

AD-A267 054



RL-TR-93-38
Final Technical Report
April 1993



KBSA CONCEPT DEMO

Center for Strategic Technology Research Andersen Consulting

Michael DeBellis, Kanth Miriyala, Sudin Bhat,
William C. Sasso, and Owen Rambow

DTIC
ELECTE
JUL 20 1993
S E D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

93-16352



Rome Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, New York

9 3 1 6 3 5 2

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

Although this report references limited documents listed below, no limited information has been extracted:

RADC-TR-88-205, Knowledge Based Requirements Assistant, October 1988, Distribution limited to U.S. Government agencies and their contractors; critical technology; Oct 88. Other requests must be referred to RL/C3AB, 525 Brooks Rd, Griffiss AFB NY 13441-4505.

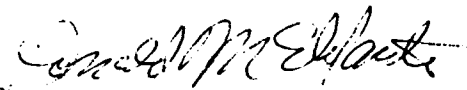
RADC-TR-89-98, KBSA Performance Estimation Assistant, August 1989, Distribution limited to U.S. Government agencies and their contractors; critical technology; Aug 89. Other requests must be referred to RL/C3AB, 525 Brooks Rd, Griffiss AFB NY 13441-4505.

RADC-TR-90-115, Analysis of Improved Many-on-Many, April 1990, Distribution limited to U.S. Government agencies only ; test and evaluation; Apr 90. Other requests must be referred to RL/C3AA, 525 Brooks Rd, Griffiss AFB NY 13441-4505.

RADC-TR-90-418, Knowledge-Based Project Management Assistant for ADA Systems, December 1990, Distribution limited to U.S. Government agencies and their contractors; critical technology; Dec 90. Other requests must be referred to RL/C3AB, 525 Brooks Rd, Griffiss AFB NY 13441-4505.

RL-TR-93-38 has been reviewed and is approved for publication.

APPROVED:


DONALD M. ELEFANTE
Project Engineer

FOR THE COMMANDER:


JOHN A. GRANIERO
Chief Scientist
Command, Control and Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE April 1993		3. REPORT TYPE AND DATES COVERED Final Sep 89 - Sep 92	
4. TITLE AND SUBTITLE KBSA CONCEPT DEMO				5. FUNDING NUMBERS C - F30602-89-C-0160 PE - 63728F PR - 2532 TA - 01 WU - 32	
6. AUTHOR(S) Michael DeBellis, Kanth Miriyala, Sudin Bhat, William C. Sasso, Owen Rambow					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Center for Strategic Technology Research Anderson Consulting 100 South Wacker Drive, E9E Chicago IL 60606				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3CA) 525 Brooks Rd Griffiss AFB NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-93-38	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Donald M. Elefante/C3CA/(315)330-3565.					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The Knowledge-Based Software Assistant (KBSA) Concept Demonstration System project had three main goals: (1) Integrate KBSA technology concepts developed up to the time of the effort into a single life cycle processing system. Also, provide additional insight to guide the direction of later KBSA contractual efforts. In particular, provide insights into the requirements for a) an integrated high-to-medium-level interface suite and b) high-to-medium-level process mediation and coordination. (2) Gain leverage from current industry research in, or related to, KBSA technology areas. (3) Provide a vehicle for technology transfer to complement the efforts of the KBSA Technology Transfer Consortium. This report describes the work accomplished under the project and presents its suggestions for future work in the area.					
14. SUBJECT TERMS Automatic programming, artificial intelligence, knowledge-based software, software engineering, systems engineering				15. NUMBER OF PAGES 150	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

1. Introduction

The Knowledge-Based Software Assistant (KBSA) Concept Demonstration System (Concept Demo) project, sponsored by Rome Laboratory, USAF, took place over three years (between September, 1989 and September, 1992).

1.1 Goals of the Concept Demo Project

The major goals of the project, as defined in section 1.3 of its Statement of Work, were these:

- Integrate KBSA technology concepts developed thus far into a single life cycle processing system. This work will provide additional insight to guide the direction of phase 2 and phase 3 KBSA contractual efforts ... particularly insights into the requirements for (a) an integrated high-to-medium-level interface suite and (b) high-to-medium-level process mediation and coordination.
- Gain leverage from current industry research in, or related to, KBSA technology areas.
- Provide a vehicle for technology transfer [to] complement the efforts of the KBSA Technology Transfer Consortium.

1.2 Structure of the Final Report

This report describes the work accomplished under the project and presents its suggestions for future work in the area. It consists of six sections:

1. Introduction (this section);
2. Overview of the Concept Demonstration System;
3. Description of the Concept Demo Architecture for Intelligent Assistance;
4. Formal Validation in the Concept Demonstration;
5. Concept Demonstration Software Development Process Model;
6. Lessons Learned and Issues Facing Future KBSA Efforts;
7. Bibliography; and
- A. Appendix with Technology Transfer Survey Feedback.

Section 2 will describe how the Andersen Consulting has drawn upon many sources, including earlier KBSA work, leading KBSE research from industry and academe, and commercial technology, practice, and insight to build the Concept Demo. Section 3 will then discuss, in greater detail, the Concept Demo's process mediation and coordination mechanism and the mixed-initiative approach used in its interface. These capabilities support and enact the process model described in section 5. Section 4 describes the evaluation of formal software validation technology undertaken in an extension to the original contract effort. In

conclusion, Section 6 will relate the current state of Concept Demo technology transfer efforts, and will present the lessons we have learned as insight for future KBSA research program efforts.

Please note that section 4 contains its own bibliography.

1.3 Acknowledgments

The authors of this report wish to acknowledge the valuable contributions made to the Concept Demo project by the following organizations and people:

- Rome Laboratory: Don Elefante, Doug White, Frank Sliwa, Nancy Roberts, Joe Carozzoni, and Lou Hoebel.
- Andersen Consulting: Bruce B. Johnson, Gilles Lafue, W. Michael Evangelist, Wojtek Kozaczynski, Chunka Mui, Gui Cabral, Stan Letovsky, Gerry Williams, Steve Wagner, Gadi Friedman, Anoop Kumar, and Inessa Lekakh.
- Information Sciences Institute: W. Lewis Johnson and Martin Feather.
- CoGenTex, Inc.: Tanya Korelsky and Owen Rambow.
- Kestrel Institute: Richard Jullig, Marilyn Daum, Xic Lia Xian, Doug Smith, and Tom Pressburger.
- Reasoning Systems, Inc.: Gordon Kotik and Scot Brooks.

and Professor Elliot Soloway of The University of Michigan.

Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification _____	
By _____	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

2. Overview of the KBSA Concept Demo System

2.1 Introduction

This section describes the architecture for the complete Concept Demo system. We wished to reuse, where possible, previous KBSA software in developing the system. The decision to reuse previous prototypes needed to be balanced against other requirements to develop a well integrated, relatively robust system. The first major design decision made in the project was what hardware/software platform to build the Concept Demo on and how to leverage previous KBSA prototypes. As part of this decision making process we acquired all the available KBSA prototypes and experimented with them and with their support environments – Software Refinery™, Socle, AP5/POPART, and the KBSA Framework. Based on these experiments, we chose Software Refinery (or Refine™ for short) on the Sun workstation as our primary development environment. The reason for this decision was that unlike other KBSA support environments, Refine was a commercial product and hence was more robust, stable, and better integrated than non-commercial KBSA environments.

Our decision to use Refine implied that we would integrate Refine-based KBSA prototypes (the PMA and Development Assistant) and reimplement requirements and specification functionality from prototypes (KBRA, the Specification Assistant, and Aries) built in other environments. Our development of the Concept Demo system can thus be divided into two types of efforts: integration and (re)implementation. Section 2.2 describes our work to integrate existing KBSA prototypes and other related tools. Section 2.3 describes the general framework and requirements/specification functionality we developed in Refine.

2.2 Integration

Figure 2-1 shows a high-level data-flow diagram (DFD) for the complete Concept Demo system. All of the process circles that have an underlined title are systems that were developed separately from the Concept Demo project. Some of these systems such as the CoGenTex text generator and the PMA have been modified and extended as part of the Concept Demo. All of the process circles that do not have underlined titles were developed as part of the Concept Demo project. The system that required the bulk of the Concept Demo development effort was the CD Framework. The CD Framework includes general functionality that spans the complete life-cycle such as a wide-spectrum specification language, process representation, and user-interface capabilities. The CD Framework is used as an integration vehicle for the entire system and as a platform on which to build the Concept Demo requirements and specification functionality. In this section we discuss the systems that have been integrated into the Concept Demo via the CD Framework.

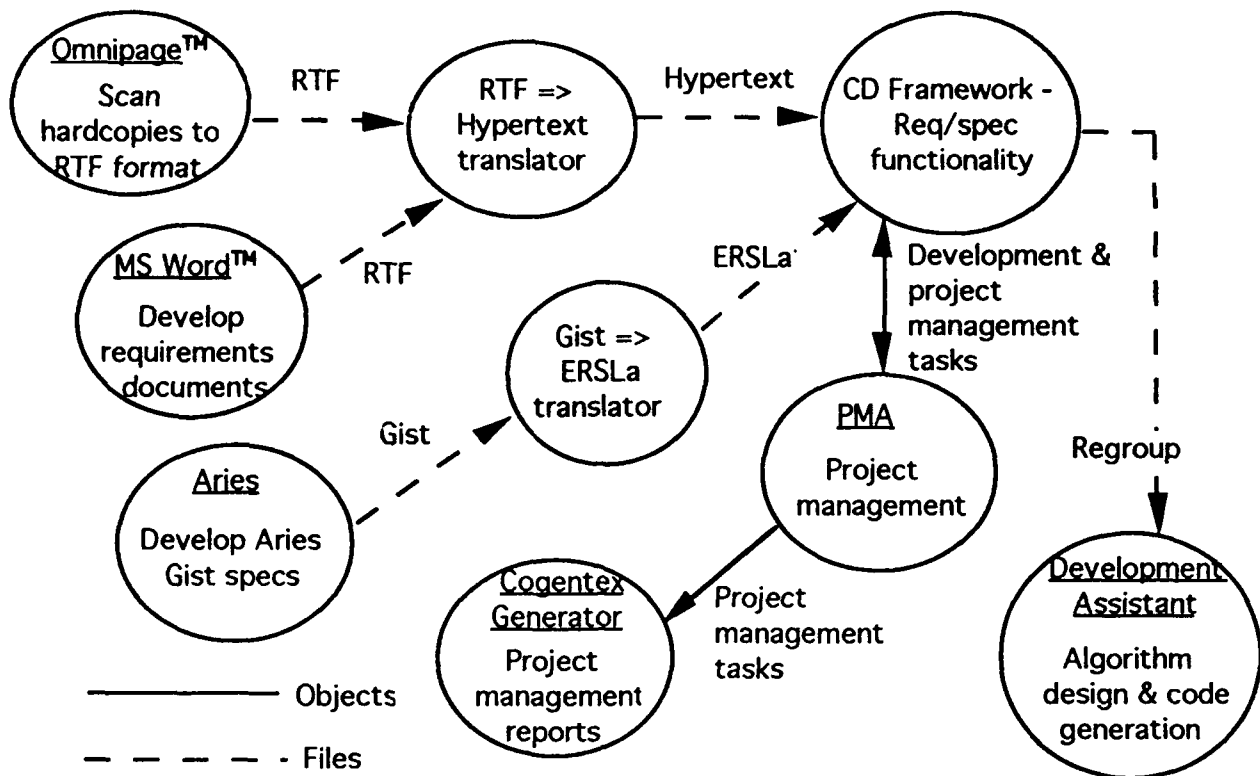


Figure 2-1: High-level DFD for the Concept Demo System

2.2.1 Incorporation of Requirements Documents

The text entry capabilities in the CD Framework are limited to using Emacs as part of the Refine environment. We wanted to also incorporate documents developed using WYSIWYG text processing systems such as Microsoft Word™ and existing large requirements documents. To incorporate such documents, we developed a translator that takes documents in Rich Text Format (RTF – a WYSIWYG standard promoted by Microsoft) and translates them into the hypertext format used in the Concept Demo. The RTF to Hypertext translator uses heuristics developed in the TAO system [Swaminathan 92] to group document paragraphs into hypertext nodes and sub-nodes based on the format of the document (e.g., use of bold face, section numbering, etc.) The translator has been used to automatically translate large documents such as the Hughes radar tracking text specification used in the Development Assistant [Smith 91] and the Concept Demo documentation [Andersen 92] into hypertext format.

2.2.2 Incorporation of Aries Gist Specifications

The Acquisition of Requirements and Incremental Elaboration of Specifications (Aries) project [Johnson 90] was a follow on to the KBRA [Harris 88] and Specification Assistant [Johnson 88a] projects. Aries developed an integrated requirements and specification environment with features from the KBRA and the Specification Assistant. Aries also explored specification reuse. We wished to

be able to utilize the large reusable specifications that were being developed as part of the Aries project. Even though Aries was built in the AP5/Popart environment and the Concept Demo was built in Refine, two developments made it possible to build a translator from Aries Gist to the Concept Demo specification language:

- 1) The Aries project developed a new version of Gist that used a syntax similar to the Refine language.
- 2) The Concept Demo developed the Extended Refine Specification Language (ERSLa) which extended Refine with constructs from Gist such as invariants and demons. ERSLa is described in detail in section 2.3.1.2.

The Gist to ERSLa translator was developed by ISI as part of their sub-contract on the Concept Demo project. It automatically translates approximately 80% of Gist specifications into ERSLa. Type definitions, unary relations, binary relations, and invariants can all be reliably translated from Aries Gist to ERSLa. Some types of demons and procedures from Gist require manual translation. The Gist to ERSLa translator was used to translate the large Aries upper model specification into ERSLa as well as several smaller specifications in the air traffic control and traffic light domains.

2.2.3 PMA Integration

As part of our requirements and specification functionality, we built an agenda mechanism to keep track of problems and suggest solutions for evolving requirements and specifications. We wanted this process guidance functionality to be tightly coupled with the project-management functionality provided by integration of the KBSA Project Management Assistant (PMA). Thus, the integration of the PMA was done at the object level, instead of at the file level. The essential PMA domain model is loaded as part of building the CD Framework. The task objects used to represent individual development steps in the CD requirements/specification environment are subclasses of the task class used by the PMA. In addition, it is possible to develop subtask hierarchies that start with PMA group level tasks and can be decomposed down to tasks that represent individual development steps such as transformations. The Concept Demo requirements/specification functionality can be built starting on a Lisp image that has the PMA loaded. This allows the full functionality of the PMA to be used on Concept Demo tasks.

The PMA has been extended by the integration of the CoGenTex text generator. This is used to automatically generate project management reports from the PMA knowledge-base. This work was done as a subcontract to the Concept Demo by CoGenTex. The text generator consists of three parts: the generator shell, developed previously by CoGenTex; the information specific to the application at hand (the generation of reports for the PMA); and the Intermediate Knowledge

Representation Scheme (IKRS), which gives the text generator access to the knowledge represented in the Concept Demo.

2.2.4 Generation of Project Management Reports

The application-independent generator shell was not altered as part of the text generation effort for the PMA. It consists of two main components:

The *realizer* handles all English-specific sentence-level processing.

The *sentence planner* determines the form of the sentences that the realizer will generate.

Furthermore, the generator shell contains code that interprets the macro-language commands used to encode application-specific information for the *text planner*, which determines content and structure of the target text, without concern for linguistic form.

2.2.4.1 The Application-Specific Generator

The first task in building a text generator for the PMA was to determine the types of text that should be generated. This task was performed at CoGenTex with close feedback from Andersen personnel, thus giving domain experts the opportunity to provide input into this crucial phase. The main constraints are the available knowledge in the underlying application program, and the requirements on the output texts. In the PMA, the knowledge representation provides a hierarchical decomposition of tasks. These tasks are annotated with semantic information that identifies them as one of a limited and predefined set of task types -- requirements, specification, issues resolution, coding, testing etc. Furthermore, each task is annotated with begin and end date. After carefully studying the available information and the usefulness of different types of text, it was decided to generate texts that for a specific period describe all ongoing activity. The text is structured according to a typical software development model: requirements analysis precedes specification, which precedes coding, etc. In addition, another model was developed to correspond with the KBSA process model [Sasso 90] used in the Concept Demo. If a task has subtasks, then the subtasks are described before the next task in the software development model is discussed. For each of these tasks, the text follows the hierarchical decomposition of tasks in the PMA knowledge representation. A limit can be set on the depth at which such task decomposition is described, in order to describe tasks at the appropriate level of detail for different users.

The generator shell needed to be augmented with three types of information:

The concepts that the texts communicate needed to be defined. Concepts are language-independent, frame-like representations of communicative content.

For the PMA, the concepts include budget-item, milestone, and requirements . Concepts are expressed in a macro-language interpreted by the generator shell.

The text planner was encoded. It accesses the knowledge base of the PMA (via the IKRS; see below) and determines structure and content of the target text. The structure and content is expressed as a stream of conceptual representations (encoded in terms of the concepts) that the text planner sends to the sentence planner. The text planner is expressed in a macro-language interpreted by the generator shell.

The conceptual dictionary was defined. It contains translations from concepts to linguistic structures, that encode lexical and deep-syntactic realization choices.

2.2.4.2 The Intermediate Knowledge Representation Scheme

The Intermediate Knowledge Representation Scheme (IKRS) is a representation of PMA domain knowledge within the text generator. It has two distinct purposes:

Certain inferences and calculations about domain facts must be made for the sake of text generation. These inferences should not be made by the text generator proper, because they are not related to communication (only motivated by it). On the other hand, the text generator should not expect the underlying application program to be modified for its purposes. The IKRS thus allows for conceptual modularity. In the PMA text generator, one of the main tasks of the IKRS is to represent and perform calculations on time points and time intervals, which is needed both for structuring the text, and for choosing proper grammatical tense for sentence realization.

The IKRS also allows for an easy integration of the text generator with the underlying application, since the interface can be concentrated in one file, rather than scattered throughout the generator. Thus the IKRS is a crucial tool in modular software development.

2.2.4.3 Sample Text

The following is a sample text illustrating the description of tasks performed using the KBSA process model in the Concept Demo.

Project Report

This report covers the period from October 11th until October 23rd.

Progress Against Milestones

Building of the Air-traffic-management started. Requirements acquisition for the Air-traffic-management was completed. The specification for it started. Outstanding issues were resolved. Manually created issues were addressed. Transferring the slots to the flight plan resolved the issue of adding a flight plan to the air-traffic-management domain model. Syntactic issues were resolved. Defining a referenced object addressed the issue of an undefined name reference. This issue was resolved manually. Resource issues were resolved. The issue of illegal access of aircraft location was addressed by splicing a communicator to resolve the illegal slot usage. This issue was resolved manually.

Cost Status

35% of the project budget has been spent (\$7,000 out of \$20,000).

Schedule Status

The project is currently on schedule.

2.2.4.4 Project Management Report Generation Summary

The CoGenTex text generator for the PMA in the Concept Demo has shown that in a relatively modest effort it was possible to build a useful tool for project managers. The principal areas for future improvement are the identification of a larger set of conceptually defined task types (so that the generator can be more explicit in describing the structure of projects) and a more complete integration with the Concept Demo knowledge-base. The text generator is the one module in the Concept Demo that is only integrated at the conceptual level. I.e., it is currently necessary to manually create an object structure that corresponds to the task hierarchy that needs to be paraphrased. To complete the actual integration would not be difficult – it would involve the definition of accessor functions to communicate between the text generator and PMA objects.

2.2.5 Integration with the Development Assistant

The Development Assistant is used in the Concept Demo to illustrate the KBSA approach to the downstream part of the development life-cycle. Integration with the Development Assistant was achieved by building the ERSLa language on top of Regroup – the specification language used by the Development Assistant. ERSLa was originally developed on top of Refine and for simplicity we often still describe it as an extension of Refine since Refine is more widely known than Regroup. Regroup specifications are all legal ERSLa specifications. This allows the requirements/specification functionality in the Concept Demo to be used to develop Regroup specifications and to link formal Regroup specifications to informal requirements in the Concept Demo hypertext system. This integration is illustrated in scenarios 3.4.6 and 3.4.7 in the Concept Demo User's Manual [Andersen 92].

2.3 Reimplementation and New Functionality

A high-level view of the data-flow for the modules of the Concept Demo built using the Concept Demo framework is shown in figure 2-2. In the following section we will discuss the Concept Demo framework and the functionality built on it.

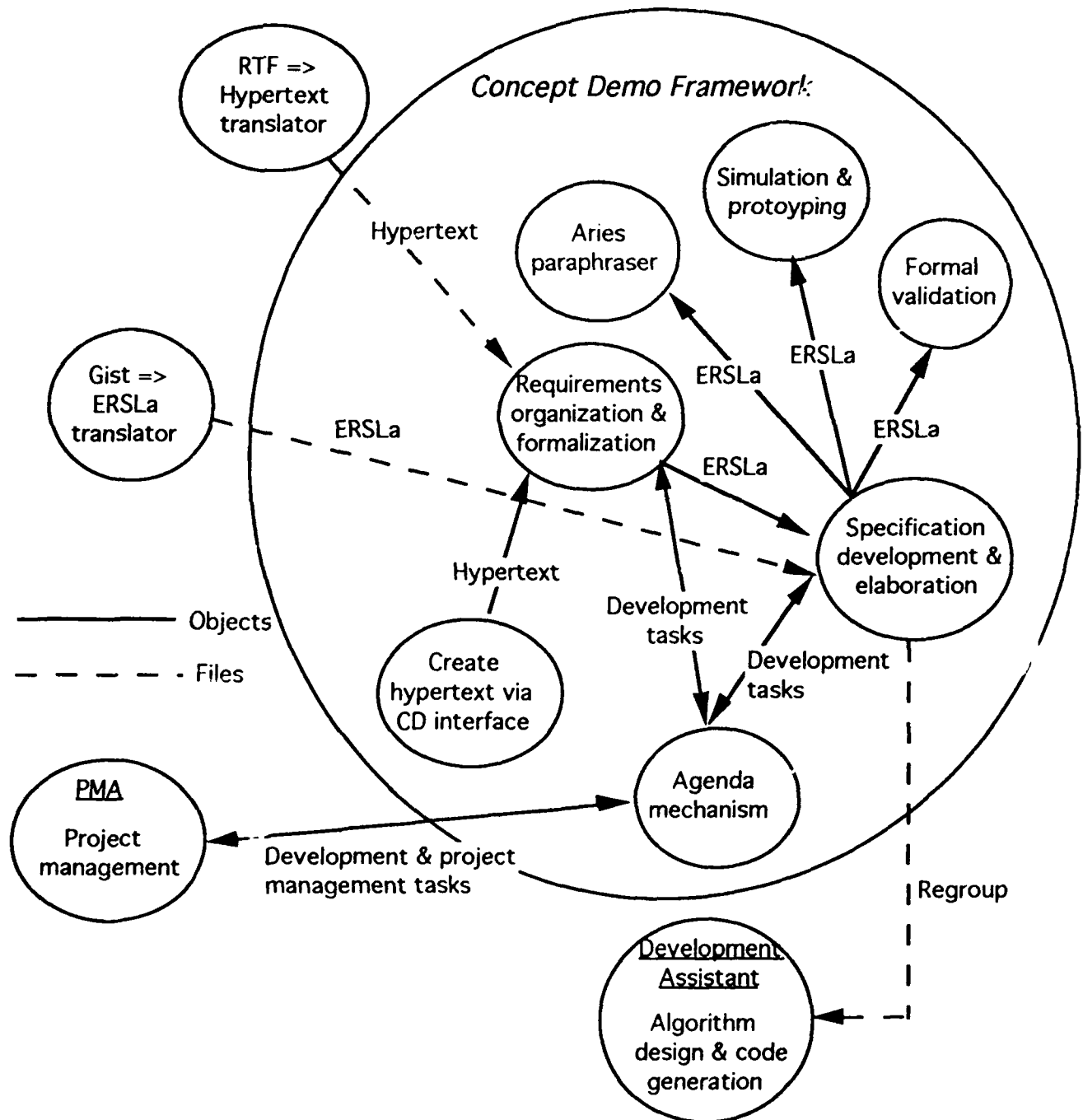


Figure 2-2: Expanded DFD Showing the Concept Demo Framework

2.3.1 Concept Demo Framework

The Concept Demo framework consists of the presentation-based interface, the ERSLa specification language, the history mechanism, and the agenda mechanism. ERSLa and the presentation-based interface are discussed in the following sections. The history and agenda mechanisms are discussed in detail in section 3 of this report.

2.3.1.1 Presentation-based Interface

Figure 2-3 shows the architecture for the Concept Demo user interface. The starting point is an X version of Common Windows – a window system that is part of the Allegro™ Lisp environment that Software Refinery is built on. The next level is Intervista™. Intervista is the window environment that is part of Software Refinery and provides UI capabilities especially suited to software development environments.

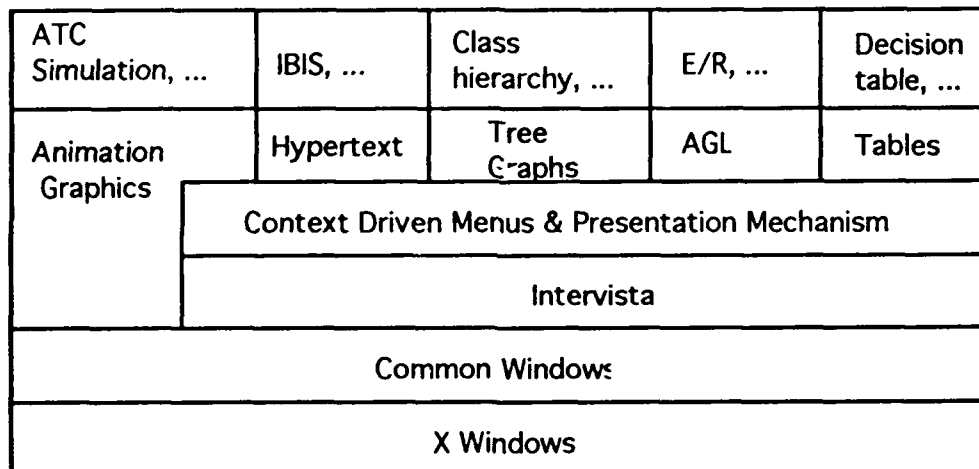


Figure 2-3: Architecture for the Presentation-based Interface

All of the levels above Intervista are additions developed in the Concept Demo. The following is an overview of these general Concept Demo user interface utilities. For details on how to use and extend these utilities see section 5.4.2 in the Concept Demo User's Manual.

Context driven menus provide general utilities for associating menus with objects and presenting specific menus based on the context of the object.

The presentation mechanism maintains consistency between the knowledge-base and the user interface.

Animation graphics are built primarily on Common Windows. These are utilities to create and animate bitmap icons.

The hypertext mechanism provides general utilities for associating strings with kb-objects (building on Intervista's mouse-sensitive printing windows) and for structuring and viewing hypertext documents.

Tree graphs extend the functionality in Intervista for graphing trees.

AGL (Aesthetic Graph Layout) is an extension of the Diagram Windows component in Intervista for visualizing general graph structures. AGL provides utilities to automatically position nodes in network graphs and to automatically route edges in such graphs in order to minimize bends and crossings. The node-positioning heuristic works moderately well for small graphs. The edge-routing algorithms are mature and work well even on large graphs.

Tables is a set of utilities that extends the "You're On Your Own" (or YOYO) table provided in the Refine 3.1 version of Intervista.

Representative user presentations are shown across the top of the diagram above the particular utility that they use. For example, the E/R presentation is built using the AGL utilities. Since the majority of these presentations deal with requirements and specifications, their detailed descriptions are provided in section 2.3.2 (below) describing requirements and specification functionality.

2.3.1.2 Extended Refine Specification Language (ERSLa)

The ERSLa language is built on top of the Regroup language used by the Development Assistant project. Regroup is itself an extension of the Refine language. For a description of the Refine language, see the Refine User's Guide. For a description of Regroup and the Development Assistant, see the Development Assistant final report [Smith 91].

There are three types of ERSLa extensions:

- 1) Gist constructs.
- 2) Object-oriented constructs.
- 3) Domain-specific constructs.

The following section describes the most important extensions in ERSLa. Section 5.2 in the Concept Demo User's manual provides extra detail for ERSLa users.

2.3.1.2.1 Gist Extensions

The ability to declare and maintain constraints is one of the most important features of the Gist specification language that does not exist in Refine. The two Gist constructs for defining such constraints are the *invariant* and the *demon*. An invariant describes some logical condition that is always meant to be true. A

demon consists of a predicate to test (the when condition) and an action to take when the predicate is true.

Using the grammar definition capabilities in the Dialect component of Software Refinery, it was relatively simple to define a new grammar that inherits from the Regroup grammar and adds constructs for invariants and demons. However, we wanted to also make these constructs executable. To do this, we built on the policy mechanism in the PMA [Daum 90]. The PMA policy mechanism allows managers to define high-level constraints on the project-management knowledge-base and actions to take when policies are violated. The policy mechanism is implemented using a general knowledge-base constraint mechanism called a monitor which is equivalent to the demon construct in Gist. Like the demon, the PMA monitor consists of a "when" part to define the policy and a "do" part to define the action taken when the policy is violated. Demons in ERSLa get translated to these PMA monitor objects. Invariants in ERSLa are transformed to monitors where the "when" condition is the negation of the invariant and the action is to report a violation and interrupt the specification simulation.

The current implementation of constraints in ERSLa is inefficient. Any demon or invariant which has a universal quantification over two or more classes will execute very slowly if there are more than a few (five or more) instances of the classes. Many of the efficiency problems inherent in PMA policies and ERSLa constraints have been addressed in further work by Xio Lia Xian (the original developer of the PMA policy mechanism) reported in [Jagadish 92].

2.3.1.2.2 Object-Oriented Constructs

KB-Modules. A kb-module is an object that collects other objects into functional modules. Unless explicitly declared, objects in one module are hidden from other modules. Use relations between modules are explicitly declared. This is an attempt to represent and reason about encapsulation at a larger level of granularity than that provided by classes. In the Concept Demo, systems are compositions of kb-modules (where each module itself may contain a number of sub-modules). Kb-modules use a protocol similar to the package protocol used in Common Lisp for describing relations between modules. Each module has exported objects. Each module has a set of modules which it uses and another set which it is used by. An object X is visible to another object Y if and only if

- 1) X and Y are in the same module, or
- 2) the module that X is contained in is used by the module that Y is contained in and X is exported from its module.

Kb-modules can contain many types of objects. They may contain formal objects describing requirements and specifications. They may contain informal (e.g., hypertext)

objects describing requirements or tasks. They may contain objects which make up the Concept Demo system. An important type of kb-module is a library module. A library module contains instances of other kb-modules. Note that there is an important difference between loading the definition for a kb-module (a particular instance of the kb-module class) and loading the objects contained in a module. Typically there will be many more modules defined than are actually loaded into the environment.

Methods. ERSLa extends Refine with a message passing protocol that is similar to that used by the Common-Lisp Object System (CLOS). A method definition in ERSLa looks exactly the same as a function definition in Refine, except that "method" is substituted for "function". In defining a method, the data-type for the first argument must be declared, and that data-type must be an object-class. The data-type for the first argument of the method is used to determine which class this method is defined for. Invocation of methods is exactly the same as invocation of functions. This is similar to CLOS, except that only the first argument of the method can be used to discriminate the handler in ERSLa.

2.3.1.2.3 Domain Specific Constructs

While developing the Concept Demo, we frequently presented the evolving ERSLa language to software development professionals for feedback. One common issue raised was the difference between a programming language and a specification language. In the research community this debate often centers around the degree of formality, declarativeness, and executability of the language. However, these issues were of little concern to the developers that we talked to. Instead, they wanted representations that allow them to describe problems easily, intuitively, and using existing notations. In addition, they did not seem to be concerned with distinction between the language and the interfaces that support it. For this reason we believe that domain specific representations — combinations of language extensions and graphic interfaces — will play an important role in the long term adoption of KBSE. The following are examples of such representations.

In the current version of ERSLa the "domains" are types of programming problems rather than application domains. In the long term, we would like to develop a hierarchy of domains, building first on a general purpose specification language, then adding programming domain constructs (such as those described below) and finally building application domain constructs. These ideas are described in more detail in [Cabral 91].

Measurement Units. In domains as diverse as scientific programming and accounting, the association of units with numbers occurs on a regular basis. The measurement-type constructs in ERSLa allow developers to associate numbers (integers or reals) with values. This provides automatic checking for entered values as well as conversions from one measurement to another — e.g., if "2

minutes" is entered where a seconds-measurement is required then it will be automatically coerced to "120 seconds".

Decision Tables. The decision table is used to describe complex decision logic. In this case, the language construct is just a place holder for the graphic interface for creating and editing the table. The graphic interface will inherit the lexical scope for the particular language construct that it is being used to edit. This means that when filling in the table, the options will be constrained by the local variables that are visible in the context of the specification object that the decision table is contained in.

Time Demons. In order to create real-time simulations to test our executable specifications, we extended the traditional notion of a Gist demon with a demon that is triggered by the passage of time. Instead of a "when" condition, time demons use the keyword "every" followed by a time measurement defined using the measurement units (e.g., "5 seconds"). The action or "do" part of the time demons is exactly the same as regular demons - i.e., any ERSLa expression.

2.3.2 Requirements & Specification Functionality

This section describes all of the requirements and specification functionality in figure 2-2 except for the agenda mechanism and the formal validation functionality. The agenda mechanism is described in section 3 of this report. The formal validation functionality is described in section 4.

2.3.2.1 Requirements Organization & Formalization

The Concept Demo uses a Hypertext presentation to enter and view informal requirements. Each requirement is entered as a text string with pointers to other objects such as the author of the requirement, other requirements with which the requirement may conflict, requirements documents that the requirement is part of, etc. The text string for the hypertext requirement is a *hyperstring*. A hyperstring and its pointers to other objects is a *hypertext node*. When a hyperstring is entered or modified, any words in the hyperstring that correspond to the names of objects describing requirements or specifications for that system are highlighted in boldface and made mouse-sensitive, providing the developer with access to context specific menus. This recognition was described with the phrase *Catch as Catch Can* in the KBRA [Harris 88], since the tool will often fail to recognize words or phrases that relate to formal objects because of the ambiguity inherent in natural language. Whenever the knowledge base is updated, part of the job of the presentation interface is to update the display of viewable hyperstrings so that references to objects that have been added or deleted will be updated.

The Concept Demo does not use natural language processing techniques, such as those found in the KBRA. It is limited to simply processing each word in the

string and looking it up in the knowledge base. However, it provides general hypertext functionality that was not present in KBRA, as well as classes of hypertext nodes and hypertext links that are geared toward requirements. These include all the classes (issues, positions and arguments) and links (responds-to, objects-to, supports, etc.) in the IBIS [Conklin 88] methodology.

2.3.2.2 Graphic Presentations

Most of the presentations in the Concept Demo relate to creating, modifying and viewing formal and informal requirements and specification objects. For that reason a detailed discussion of the "user level" presentations mentioned in section 2.3.1.1 was postponed for this section. The following conventions are used to show the relation between the user level presentations and the general interface capabilities described in 2.3.1.1.

- Presentations ending in "diagram" are built on the AGL layer.
- Presentations ending in "tree" are built on the tree graph¹ layer.
- Presentations ending in "display" are built on the hypertext layer.
- Presentations ending in "animation" are built on the animation layer.
- Presentations ending in "table" are build on the tables layer.

Presentations for class and slot definitions.

- Entity Relation (E/R) Diagram. Allows the developer to view and edit class definitions using an E/R formalism. Classes are treated as entities, slots with a range that is not a KB class (e.g., integer, string, symbol, ...) are treated as attributes, slots with a class range are treated as relations.
- Class Tree. Graphs the subclass links between a group of classes.

Presentations for instances. These presentations are useful for viewing instances and slot values. Since elements of specifications are also instances, e.g., a class definition is itself an instance of the Class (meta) class, these presentations are useful for viewing specification and requirements objects and the relations between them. These presentations are also useful for viewing objects created as a result of executing simulations.

¹ Many of the tree-based presentations are graphs that may not always be strictly trees (e.g., the function calling graph). The tree graphing utilities will still work in such cases.

- **Object Frame Display.** Displays slots for a particular object and the values for the slots. The slots and the slot values are mouse-sensitive, providing the developer with access to context specific menus. The developer can control which slots will be seen for instances of each class.
- **Edit Object Display.** Similar to the Object Frame Display except that the slot values can be edited.
- **Object Values Table.** Takes a set of instances and a set of slots and displays the slot values for each slot on each instance in an editable form. This presentation is written so that it can be easily customized to display and edit specialized views of the knowledge-base.
- **Non-functional Requirements Table.** The ERSLa meta-model includes slots that represent non-functional requirements for specification objects (e.g., required execution time for a function). These are defined using the measurement types in ERSLa. This presentation is a specialization of the Object Values Table for editing these slots.
- **Semantic Net Diagram.** Takes a group of instances and a group of relations and graphs the links between those instances. This presentation is written so that it can be easily customized to display and edit specialized views of the knowledge-base.
- **IBIS Diagram.** A customization of the Semantic Net for displaying IBIS instances (positions, issues, arguments) and links (responds-to, supports, etc.).
- **REMAP Diagram.** A customization of the Semantic Net for displaying objects and links corresponding to portions of the REMAP model for requirements tracking described in [Ramesh 91].
- **Hypertext Display.** Integrates the Object Frame display with a mechanism for dynamically creating mouse sensitive highlighting for references to objects in hyperstrings.
- **ATC Animation.** Has bitmaps defined for aircraft and airports, and uses time-demons to update the location of an aircraft on the screen when it takes-off, lands, or changes its position.
- **Traffic Light Animation.** Has bitmaps (cars, trucks, and colored stop lights) to simulate the traffic light specification. This presentation has not been integrated with the traffic light specification in the current version.

Process req/spec presentations. These are useful for viewing control and data-flow views of requirements and specifications.

- **Information Flow Diagram.** Provides information on how the slots of a class are accessed and modified by functions and other process oriented specification constructs.
- **Syntax Based Display.** Displays the ERSLa description for a KB object and performs mouse sensitive highlighting based on the parse tree of the object.
- **Function Decomposition Tree.** Takes a function and graphs all the functions called by that function (and all their callers, etc.)
- **Decision Table.** The graphic interface for editing the decision table construct in ERSLa.

KB Module presentations.

- **Used-by Relations Tree.** Displays the use relations between knowledge base modules.
- **Module Decomposition Tree.** Takes a starting kb-module and shows all the kb-modules which are contained in that module (as well as any modules contained in those modules, etc.) This is analogous to the top-down product decomposition diagrams in 2167a Software Design Documents.

Software process presentations.

- **Task History Tree.** Graphs the actions (instances of tasks) performed in a development session.
- **Task Decomposition Tree.** Graphs all the subtasks of a task.

2.3.2.3 Specification Feedback

One advantage of a formal specification is the ability to reason about the specification and provide feedback to the developer. The Concept Demo provides four types of specification feedback: Syntactic Analysis, Resource Analysis, Paraphrasing, and Simulation.

- **Syntactic Analysis** provides feedback on the syntactic correctness of each specification construct. This includes problems such as type inconsistencies, references to undefined objects, and parameter mismatches.

- **Resource Analysis** is used to analyze the resource usage of objects across kb-module boundaries. Objects that violate the encapsulation rules described in section 2.3.1.2.2 are flagged with issues.
- **Paraphrasing** is used to translate formal specification constructs into English. The Concept Demo paraphraser is adapted from the software used by the Aries system to paraphrase Gist specifications. See [Williams 90] for a detailed description of the paraphraser.
- **Simulation** is used to evaluate the behavior of an executable simulation. During simulation any invariant violation interrupts the simulation and is recorded in a log for the simulation. The animation interfaces described above provide graphic capabilities for controlling and viewing simulations. The ERSLa demons triggered by the passage of time provide high-level constructs to create and control simulations.

2.3.2.4 Evolution Transformations

The Specification Assistant and Aries projects developed the concept of *Evolution Transformations* for performing "stereotypical, meaningful changes to the specification" [Johnson 88] in "systematic, controlled ways" [Johnson 90b]. Evolution transformations are performed with the history mechanism, allowing a developer to explore alternative development options and to undo and redo any sequence of development steps. An important accomplishment of the Concept Demo was to integrate evolution transformations with the requirements and specification feedback discussed above using a task based process formalism. The architecture for this integration is discussed in detail in section 3. Here we provide a description of the types of evolution transformations that exist in the Concept Demo.

Evolution transformations in the Concept Demo can be categorized along two orthogonal dimensions:

- 1) The degree of automation/sophistication provided by the transformation.
- 2) The goal that the transformation is trying to achieve.

Along the first dimension (automation/sophistication), the Concept Demo transformations fall into three different categories from least to most sophisticated:

- 1) **Tool wrapping transformations.** These transformations are really programmatic ways of invoking other tools such as editors or functions to compile and load files. These transformations provide functionality similar to "derivars" in the Artifacts system [Karr 89] or the tools wrapped by envelopes in the Marvel [Kaiser 88a] system. Examples of these types of transformations are

invoking Emacs to define or edit an informal requirement and loading a kb-module from a file.

2) Basic KB manipulation transformations. These transformations provide functionality to create and make basic changes to specification language objects. They are similar to the functionality provided by sophisticated object-oriented development environments such as KEE [Fikes 85] to create and modify classes, slots, and methods. These transformations are also the building blocks for the third group of transformations. Examples of these types of transformations are defining new classes and adding slots to an existing class.

3) Pattern matching transformations. These are the truly powerful evolution transformations. The previous two groups of transformations for the most part simply add to or make local changes to the knowledge-base. These transformations use the pattern matching capability in the Refine transformation engine to make wide-scale changes to the specification and evolve it in a controlled fashion. An example of this type of transformation is the *bundle* transformation. Bundle moves existing slots from one class to another while maintaining consistency with existing functions, invariants, etc.

Considering the transformation's goal (the second dimension), the Concept Demo transformations can be grouped into the following 5 categories:

1) Project-management. These transformations work on process objects. Examples of these transformations are assigning a task to a developer and creating a task structure for the development of a kb-module. Most of these transformations are basic KB manipulation transformations.

2) Requirements definition. These transformations are for creating new hypertext requirements via Emacs or for translating existing RTF files into hypertext format. Most of these are tool wrapping transformations.

3) Requirements elaboration. These transformations edit and annotate existing requirements. These are a combination of tool wrapping and KB manipulation transformations. Examples of these transformations are adding IBIS nodes and links to a hypertext requirements document.

4) Specification definition. These transformations create initial specification language definitions. They are a combination of tool wrapping and basic KB manipulation transformations. Examples of these transformations are creating classes, invariants, etc.

5) Specification elaboration. These transformations evolve existing specifications. They are primarily pattern matching transformations. Examples of these transformations are bundle and splice communicator.

Scenarios 3.4.2, 3.4.3, and 3.4.4 in the Concept Demo User's Manual demonstrate the use of many evolution transformations. Section 5.3 in the manual contains a detailed description of the core evolution transformations.

3. The KBSA Concept Demo Architecture for Intelligent Assistance

3.1 Introduction

In this section we describe the Concept Demo architecture for intelligent assistance. Our primary design goal for this architecture was to enable constructing a tool using current knowledge-base, mature KBSA, and CASE technology that would provide dramatic improvements over existing CASE tools while at the same time providing a foundation to integrate future mature results from the KBSA program. Such a tool would gradually evolve into a product-level Knowledge-Based Software Assistant equivalent to the description in the original KBSA vision [Green 83]. We attempted to build the Concept Demo as a prototype of such a tool. We see the Concept Demo architecture for intelligent assistance as a specification for the architecture of such a tool.

We first describe (section 3.2) the structure of the Concept Demo knowledge-base relevant to intelligent assistance then (section 3.3) we describe the flow of objects and messages that occurs in the execution of a task in the Concept Demo. The discussions in sections 3.2 and 3.3 are at a conceptual and somewhat idealized level. In sections 3.4 and 3.5 we discuss the current implementation of this architecture. First (section 3.4) we relate what users see during the execution of example tasks from scenarios in the Concept Demo User's Manual to the task execution described in section 3.3. We then (section 3.5) describe how each task execution step is currently implemented. This discussion in section 3.5 is at a level of detail appropriate for developers who wish to understand and extend the Concept Demo system. Other readers are urged to skim or skip this section. Finally, in section 3.6, we describe how the Concept Demo architecture could be used to build a knowledge-based CASE tool that could evolve into an advanced KBSA.

3.2 The Concept Demo Knowledge-Base

Figure 3-1 shows the structure of the Concept Demo knowledge-base. Each box in Figure 3-1 is a kb-module – a collection of related classes, methods, rules, etc. For each kb-module, we show a representative example of a component from the Concept Demo that is included in that module. The arrows in Figure 3-1 show the general flow of objects and messages between modules. For example, process model objects are sent to the presentation-based interface in order to be viewed and accessed by the user, but process model objects never directly use or modify interface objects.

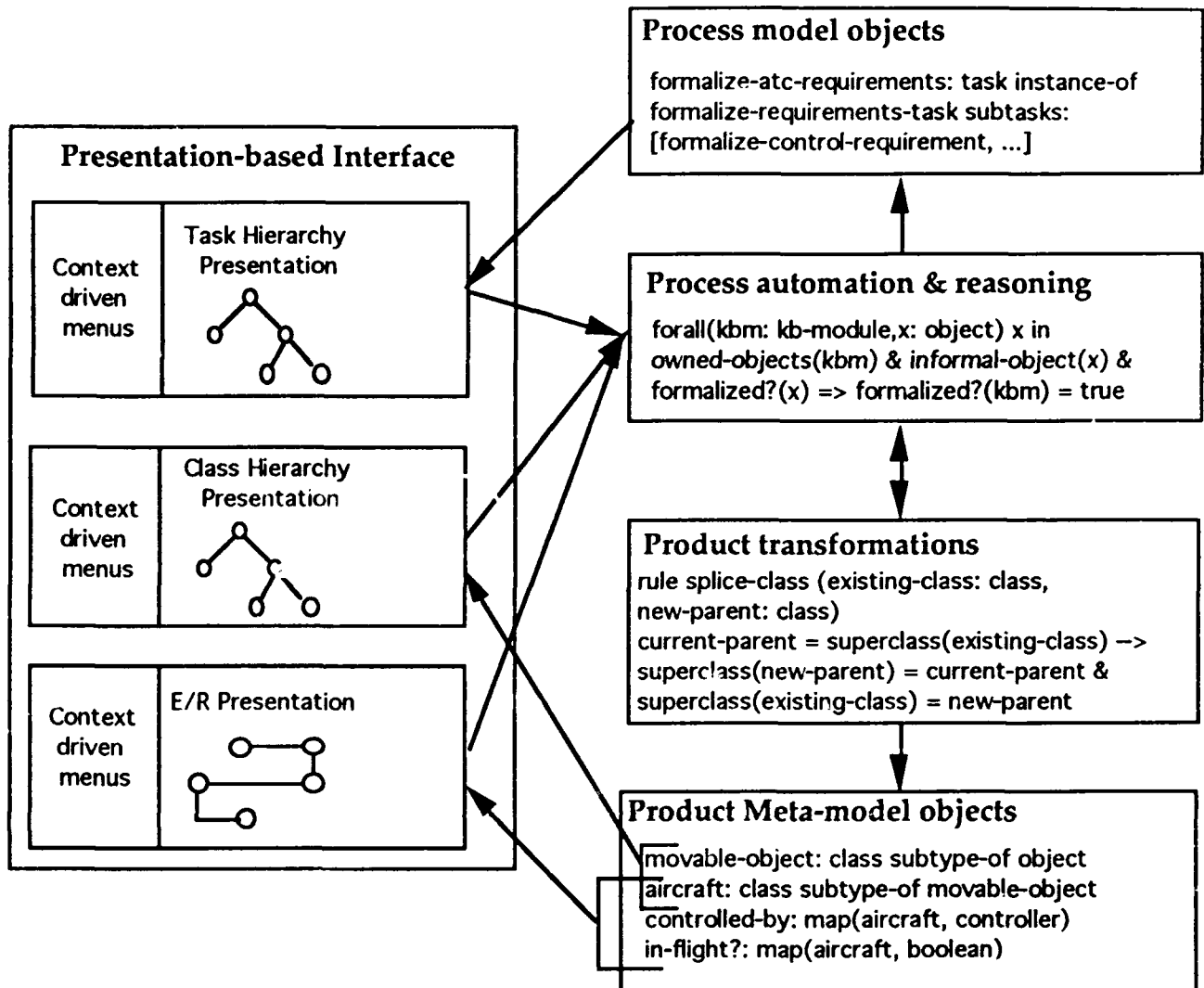


Figure 3-1: KB-Modules in the KBSA Concept Demo

The following describes these Concept Demo kb-modules.

The Presentation-based interface extends concepts first used in the KBRA [Harris 88] to provide various views of the knowledge-base that are maintained independently of the functions that modify the knowledge-base. The Concept Demo incorporates the CLF [ISI 86] technique of context driven menus into the presentation-based interface. Each menu item includes a state description that can be used to describe when it is appropriate to display the item. This allows menus to be associated with object classes in a way that is independent of interface presentations – the presentation that an object is being viewed in is simply one more part of the state description passed to the menu creation method.

The Concept Demo also uses the presentation-based interface to view the complete KB – both process and product objects. This is an extension over previous KBSA systems which used the presentation-based interface to view

only product objects. It is possible to invoke tasks both from product and process representations. This is indicated by the fact that input arrows come into the process automation box from both types of presentations.

Process model objects are classes and instances such as tasks, states, problem descriptions, etc.

Process automation & reasoning is executed by rules, methods, and functions. Examples of the reasoning executed by this kb-module includes determining the status of tasks and products, posting descriptions of problems, suggesting resolutions to problems, prioritizing outstanding tasks, and recording a development history.

All actions performed by the user are invoked through the process automation & reasoning module. Transformations are never directly invoked. Instead, they are executed as a result of invoking tasks. This provides context and rationale for the recorded use of transformations.

Product transformations evolve the software products (requirements, specifications, code, ...) being developed. In this case we use "transformation" in a much broader sense than is often used in KBSA literature. In the Concept Demo, transformations can be invocations of tools such as editors or (the more standard definition) forward chaining transformation rules specifically designed to modify software products.

Product Meta-model objects are instances of classes in meta-models that describe the requirements, specification, and programming languages used to develop products. A meta-model is a group of classes that describe the constructs in a language. It is a "meta" model because the language itself is used to model a domain or problem. The meta-model is an extension of the CLOS concept of a meta-class – a class that classes themselves are instances of. In KBSA environments all language constructs (e.g., invariants, demons, functions, classes) are instances of a corresponding class in the meta-model. KBSA technology such as the DialectTM component in Software RefineryTM or POPART [Johnson 88b] facilitate the use of a meta-model by generating parsers that create instances of object classes as a by-product of parsing.

3.3 Task Execution in the Concept Demo

In this section we will describe the execution of tasks in the Concept Demo. Before describing task execution in detail, it is necessary to describe the Task class and terminology associated with it.

3.3.1 Description of the Task Class

Preconditions are logical predicates that describe the state that must hold in order to execute a task. **Postconditions** describe the state that will hold after a task has successfully completed.

Supertask slots point from a task to its parents in the subtask hierarchy. **Subtask slots** point to its children in the hierarchy. Tasks can be decomposed from very high-level tasks that are worked on by several developers such as "Build System" to intermediate subtasks such as "Formalize Requirements" that are important steps in the process model but are small enough that they can be worked on by a single developer to subtasks such as "Formalize Control Requirement" that are instances of individual transformations and are executed in order to reach the goals of higher level supertasks.

Atomic tasks are leaf nodes in the subtask hierarchy – i.e., tasks that have no subtasks.

Transformation rules vs. Tool invocations. Atomic tasks are divided into two basic categories: tool invocation tasks and transformation rule tasks. Tool invocation tasks are executed by calling tools such as editors and compilers. Transformation rule tasks are executed by invoking meaning preserving or evolution transformations.

The Task body is a function used to execute a task. All atomic tasks must have a task body. The task body can be a call to a transformation rule or an invocation of an external tool such as an editor.

Task parameters define the inputs to a task. In order for a task to be executed its parameters must be fully instantiated. The task is then recorded in a development history and the parameters can be used to replay the task at a later date. It is also possible to store tasks that have not yet been executed (e.g., pending tasks on the agenda) with parameter lists partially instantiated. This provides intelligent assistance by automatically supplying arguments that otherwise would need to be input by the user.

3.3.2 Detailed Description of Task Execution

We now describe the steps involved in executing a task in the Concept Demo. These steps are illustrated in Figure 3-2. The processes **Invoke task** and **Update presentations** are drawn in bold because they are the start and end steps.

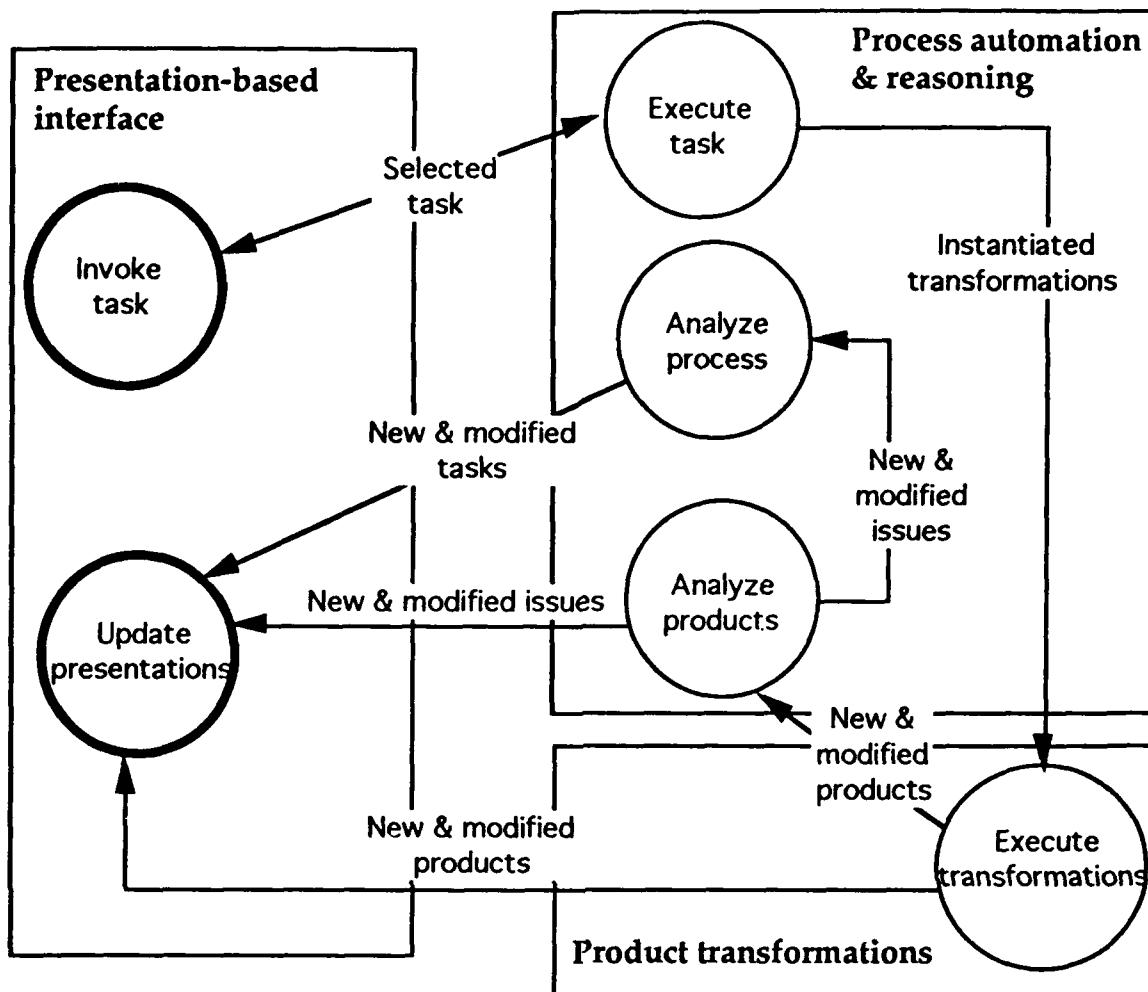


Figure 3-2: Data-flow Diagram for Execution of a Concept Demo Task

Invoke task. In this step the user invokes a task from a context driven menu. The task can be invoked by clicking on it directly via a process representation (e.g., a subtask hierarchy graph or an agenda display) or implicitly by selecting an action on a product (e.g., selecting an informal requirement and choosing the action "formalize requirement"). The user supplies any needed parameters to the task via a general "Accept Values" mechanism similar to that used in the Symbolics Genera™ environment. This guarantees that only legal objects will be passed to the task as arguments and also helps make the declaration of tasks independent of the user interface.

Execute task. Various types of reasoning may be required in order to execute a task. For example, if the preconditions of a task are not satisfied, backward chaining could be used to reason backward to tasks whose postconditions will satisfy the preconditions of the goal task. There is a two way arrow between execute task and invoke task since the execution of a task may result in the invocation of lower level subtasks which will then require parameters to be instantiated via the user interface. Execution for atomic tasks consists of simply invoking the task's body function with the task arguments.

The current implementation of the Concept Demo performs relatively shallow reasoning during task execution (this is described in detail in section 5.2). The architecture of the Concept Demo is structured so that more sophisticated reasoning such as that performed in the Marvel system [Kaiser 88] can be incorporated in the future.

Execute transformations. The result of executing the task will be an ordered sequence of one or more transformations to execute. These could be specification evolution transformations, meaning preserving transformations, or invocations of tools such as text editors.

Analyze products. After the tasks have been executed, the products which were changed by them are analyzed. This analysis is based on the process model and may include more global analysis — for example, to determine the state of the developer's workspace.

Analyze process. Product analysis will create problem descriptions called issues. Process analysis will then analyze these issues and generate possible resolution tasks. In addition, the general state of the developer's workspace with respect to the process model will be re-evaluated. This will include removing issues that have been resolved as a result of the task execution, removal of tasks that were created to resolve issues that have been removed, and propagating the effects of the completion of low-level tasks back up to the appropriate supertasks.

Update presentations. The final step in the execution of a task is to update the presentation-based interface for all modified process and product objects.

3.3.3 Saving of State and Task Information

All of the changes introduced to the knowledge-base as a result of the execution of a task are saved in a KB state. This allows the developer to easily undo or redo the effects of a task. This represents an enhancement to the history capability in CLF, since CLF will only undo/redo changes made by a development step to AP5 relations. There are often significant changes made by a development step in the CLF environment (for example, to the user interface) that are not captured in AP5. Since all significant changes in the Concept Demo are captured in the Refine KB, the developer's workspace moves to exactly the same state when a development step is undone or redone as originally resulted from the execution of the development task.

In addition to saving the state that results from execution of a task, the task itself is stored along with the parameters that it was invoked with and other relevant information (such as who executed the task, when it completed, which objects it modified, which issues it resolved, etc).

Saving the task parameters allows the task to be replayed in a different context if needed. For example, to undo a task T1, execute a new task T2, and then redo T1; it is necessary to replay T1 as opposed to simply moving to the state that resulted from T1 since moving to the state would negate the effect of T2. This replay mechanism is similar to replay in the Specification Assistant [Johnson 88a].

3.4 Examples from the Concept Demo

In this section we describe the execution of three example tasks in the Concept Demo. This section is meant to be used in conjunction with the scenario: "3.4.3 Process Guidance in the Acquisition of Informal Requirements" in the Concept Demo User's Manual. We encourage readers not familiar with this scenario to go through it ("hands on" if possible). Alternatively, they may wish to skim or skip this section.

3.4.1 Create a New Informal Requirement

This task corresponds to scenario element "3.4.3.2 Creating an Informal Requirement in Hypertext."

3.4.1.1 Invoke task. To begin, the developer selects the task of adding a new informal object from a product representation – a Semantic Net graph that currently shows requirements traceability links. There are two arguments required for the task of adding a new informal object. The first is the class that the informal object will be an instance of and the second is the kb-module that the informal object will be a component of. The first argument is input by the user from a menu of possible subclasses of informal object and the second is provided implicitly from the kb-module that the semantic net presentation is viewing.

3.4.1.2 Execute task. This is a simple task, requiring no decomposition. Its execution merely calls the function in the task body.

3.4.1.3 Execute transformations. This is a tool invocation task. The execution of the function in the body of the task invokes Emacs and allows the developer to create a textual description for the informal requirement, in this simple example the string: "All aircraft that are in-flight must be controlled."

3.4.1.4 Analyze products. Product analysis initiated after completion of the task reveals that there is a new informal requirement without a corresponding formal definition. This generates an issue requiring the creation of a formal definition for the requirement.

3.4.1.5 Analyze process. Process analysis detects the formalization issue and generates a task for creation of a formal definition for the requirement. This task is then placed on the developer's agenda. Note the message: "Adding

FORMALIZE CONTROL REQUIREMENT to agenda" in the Output Window. The agenda is prioritized and this new task is determined to be the most important task to accomplish.

3.4.1.6 Update presentations. For this task, updating of the presentation-based interface adds the informal requirement to the Semantic Net graph. At the end of presentation update, all presentations for objects that have been changed or added as a result of the task are highlighted in reverse-video. Thus, for this task the icon for the new control requirement hypertext node is highlighted.

At the end of the task (as for every task) the changes to the knowledge-base are saved in a state that can be moved to or from. In addition, the task is placed in an appropriate position in the subtask hierarchy, and relevant information about it (such as what objects it modified) is saved. The message "Knowledge-base is now in state resulting from ADDED REQUIREMENT CONTROL REQUIREMENT" is displayed in the Output Window. In this case "ADDED REQUIREMENT CONTROL REQUIREMENT" is the name of the task that was just completed. Note that the display for this task in the Output Window is in bold which means it is mouse sensitive and could be displayed, browsed, etc.

3.4.2 Formalize the Requirement

This task corresponds to scenario element "3.4.3.3 Accessing KBSA's Process Guidance Capability."

3.4.2.1 Invoke task. While the previous task was invoked from a product presentation, this task is invoked from a process presentation. The developer inquires "What should I do next?" from the "Feedback and Guidance" icon in the control panel in the upper left corner of the interface. In step 3.4.1.6 the formalization task was created and placed at the top of the agenda. This task is now displayed in a mouse sensitive format. This enables the developer to mouse directly on the task and invoke it.

3.4.2.2 Execute task. In executing this task, the Concept Demo finds two atomic tasks that could satisfy the goal of formalizing the requirement: (1) creation of a new formal definition for the requirement and (2) linkage of the requirement to an existing formal definition. The system prompts the user to select which of these two alternative tasks to use. In this scenario there is no existing formal definition that can be used, so the user selects the task to create a new formal definition. This causes the system to invoke the parameters for the "Add formal definition" task. The first parameter is the class of the new formal object to add (see the discussion of the product meta-model in section 3.2). Once the class of the formal object has been chosen (in this case, an invariant) a form is presented to fill in the name, parameters, and body of the invariant.

3.4.2.3 Execute transformations. This is a KB manipulation transformation that adds the new definition for the invariant to the knowledge-base. It also adds a "formalized-by" link from the informal requirement to the invariant and a "formalizes" link from the invariant to the informal requirement.

3.4.2.4 Analyze products. The issue describing the problem of an unformalized requirement is marked as resolved. In creating the invariant the body of the invariant was defined to be: "in-flight(ac) => controlled(ac)" – the invariant parameter list: "ac: aircraft" binds ac to stand for all instances of the aircraft class. This is the ERSLa way of expressing "If an aircraft is in-flight then it must be controlled." However, while there is a formal definition for "in-flight" in the knowledge-base, there is currently no definition for "controlled". The syntactic analysis for specification completeness detects this problem and creates a new issue describing the need to create a "map" named controlled. Because of the context in which controlled was used, the system is able to determine that controlled must be a map (a mapping from the domain of aircraft to a Boolean of true or false) – which means it must be an instance of the meta-model class for a function, method, or slot. The other formal objects do not have parameters or do not return a value — such as invariants, classes, and demons — are ruled out.

An issue is also created noting the need to "maintain" the new invariant. Invariants are a formal description of what the system must do, but they provide no description of how the system should do it. In the process model used in this version of the Concept Demo, invariants may be maintained before the complete set of requirements have been described. This is an iterative spiral approach to development. Another (waterfall like) approach would be to require all invariants to be created before any maintenance actions are defined.

3.4.2.5 Analyze process. The task to formalize the control requirement is marked as completed. New tasks are created to resolve the issue dealing with the undefined map named "controlled" and to maintain the invariant. Note the messages in the Output Window: "Adding RESOLVE REFERENCE TO UNDEFINED MAP NAMED CONTROLLED to agenda" and "Adding MAINTAIN CONTROL INVARIANT to the agenda." The agenda is prioritized and the undefined map task is determined to be the most important task to accomplish.

3.4.2.6 Update presentations. The invariant is added to the Semantic Net presentation and the "formalizes" link from the invariant to the informal hypertext version of the control requirement is drawn.

3.4.3 Resolve a Problem with the Completeness of the Specification This task corresponds to scenario element "3.4.3.4 Intelligent Assistance in the Requirements Acquisition Process."

3.4.3.1 Invoke task. This task is invoked similarly to the previous one – via a process-based presentation accessed from the control panel. Note that the invocation of the task is greatly simplified by the fact that most of the parameters (e.g., the name and domain type for the map) have already been stored on the task as part of the intelligent assistance provided by the tool. All these parameters can be determined by the context of the reference to “controlled” in the invariant. the user need only tell the system the type of map (a function, method, or slot) and the range type. The range type can also be inferred, but this capability hasn't been implemented.

3.4.3.2 Execute task. The execution of this task is simple, since there is only one atomic task capable of resolving the issue.

3.4.3.3 Execute transformations. The transformation executed for the task is a KB manipulation transformation adding the new slot to the knowledge-base.

3.4.3.4 Analyze products. The incompleteness issue regarding the control invariant is marked as resolved. In addition, there was another issue regarding a function that referenced controlled. This issue is also marked as resolved. Note the Output Window messages: “Resolving issue UNKNOWN-NAME-ISSUE-1” and “Resolving issue UNKNOWN-SLOT-OR-FUNCTION-ISSUE-1.” The issues are marked as being resolved by the currently executing task.

3.4.3.5 Analyze process. The task for creating the formal definition for controlled is removed from the agenda. Note the Output Window message: “Removing task RESOLVE UNDEFINED REFERENCE TO CONTROLLED from the agenda.” The agenda is prioritized bringing the task to maintain the invariant to the top.

3.4.3.6 Update presentations. There are no presentations visible in the scenario that are appropriate for viewing the new attribute so the only interface update is the description of the new attribute in the Output Window.

3.4.4 Task Recording

It is possible to step back and see how the tasks that we have executed have been recorded in the Concept Demo. If we select “Graph History” from the “History” icon in the Control Panel, we will see a (single-branch) tree graph of 10 nodes. Each node represents an atomic task that was executed in the current workspace. The last three nodes in the tree: “Added Requirement Control Requirement”, “Added Formal Definition for Control Requirement”, and “Defined Map Named Controlled” are the three tasks that we just executed. The other nodes are tasks that were executed before the demo image was saved to set up the workspace for the demo. We could replay any task or undo the effects of the tasks that came after a task by moving to the state that resulted from the task. If we were to move to a state that resulted from a previous task, the knowledge-base and the

presentation-based interface would be updated. If we were to proceed from that state, the development history graph would acquire another branch.

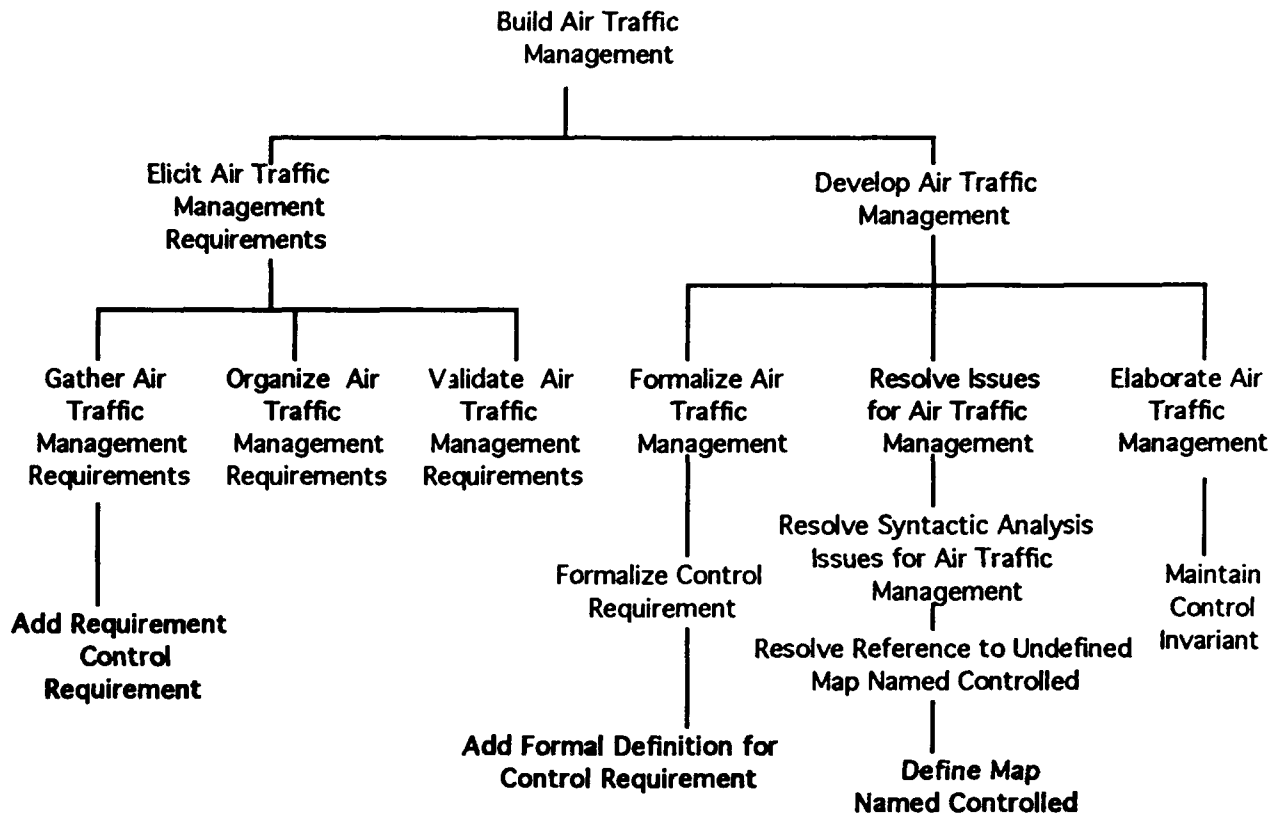


Figure 3-3: Subtask Hierarchy for Example Tasks

In addition to viewing the atomic tasks, we can see the context for these tasks in relation to higher level supertasks. This can be accomplished by clicking left on the kb-module that the objects we have been working on are a part of ("Air Traffic Management") in the "KB Modules: Used by Relations" window and selecting "Graph Task Hierarchy".

Selected portions of the task hierarchy, relevant to the three example tasks discussed above, are shown in Figure 3-3. The tasks discussed above are highlighted in bold. Figure 3-3 illustrates how the tasks we have been executing are automatically incorporated as part of higher level supertasks. This provides context for understanding the rationale behind the tasks. It also provides opportunities for propagating the effects of individual tasks back up to group level tasks for project management and activity coordination. The high-level tasks in figure 3-3 (e.g., Build Air Traffic Management, Elicit Air Traffic Management Requirements) are instances of the Project Management Assistant task class. All legal project management operations (assign a budget, mark a milestone as completed, etc.) can be performed on these high-level tasks.

3.5 Implementation of the Concept Demo Architecture

This section describes the implementation of the architecture presented above. It will be of interest only to readers who want to maintain or extend the Concept Demo. Other readers are urged to skim or skip this section.

Certain parts of the architecture are described in detail in section "5.4 Implementation Details" in the Concept Demo User's Manual. In these cases only references to the appropriate sections in the manual are given here. Section 5 of the User's Manual also contains additional information that is relevant to the following material such as a detailed description of the Extended Refine Specification Language (ERSLa).

3.5.1 Invoke task

Invoking tasks in the Concept Demo is done through the context-driven menu system. This system is explained in detail in sections 5.4.2.1 and 5.4.2.2 in the Concept Demo User's Manual.

3.5.2 Execute task

Execution of a task is accomplished by sending the task instance an "execute-task" message defined using the ERSLa method construct. This does not utilize any inferencing. Instead it relies on declarative knowledge about the kinds of tasks that can solve particular problems. This information is stored in the "possible-super-task-classes" slot for each task class.

The protocol for executing a task is to first determine if the task is atomic or not. If the task is atomic and all its parameters are instantiated, it is sent an "apply-task" message. If it is an atomic task and it requires parameters from the user, then it is sent an "interactive-execute-task" message. Interactive-execute-task will utilize task parameters that have been provided as a result of intelligent assistance and will prompt the user for any remaining ones.

If a task is not atomic, it is decomposed according to the value of its "Task-Decomposition-Method" slot (this is an example where multiple inheritance would have been useful). If the value is "exclusive-or-decomposition" then the task has two or more subtasks, either of which could satisfy its goal. The system will put up a menu with the possible subtasks and prompt the user to select one. The selected subtask will be sent an execute-task message. If the task-decomposition-method is "sequential-decomposition", then the task has a sequence of subtasks which must be executed in order and each task in the sequence is sent an execute-task message. There are other defined types of decomposition such as "parallel" that have not been completely implemented.

The files for tasks and task execution in the Concept Demo are all under the ../concept-demo/main directory. The relevant files are cd-tasks, cd-tasks-hlec,

cd-tasks-lisp, cd-task-classes, basic-cd-task-methods, cd-task-methods, issue-resolution-tasks, and set-subtask-classes.

3.5.3 Execute transformations.

All transformation¹ functions end in "-hlec"². These functions are defined in files under the ../concept-demo/main directory that begin with "hlec-". The transformation functions and the atomic tasks for those functions are generated automatically. This is done using a simple grammar called the "hlec grammar." In each file that contains transformation functions, there will be definitions at the end of the file in the hlec grammar. The hlec grammar defines how to name instances of the task (using its parameters), the name for the transformation function, text patterns used to describe the task (also using its parameters), and the superclass for the task class that will be created.

For example, the bundle transformation is defined in the file "hlec-bundle.re". At the bottom of the file is the following definition:

```
!! in-grammar('hlec-grammar)

hlec bundle-template
  context-name-pattern ["Transferred slots to", "1"]
  context-description-pattern ["Bundled", "0", "creating new class",
                              "1", "and new relations", "2"]
```

This will create a template that will be used to create the task class and the transformation function for the Bundle task. As one loads the Concept Demo, the function "cd::generate-hlecs" is called. It will generate task classes and transformation functions for all instances of the hlec class. Generate-hlecs can also be called with a keyword argument during development to regenerate new or changed transformation functions. For all instances created with the hlec grammar, generate-hlecs will call the function generate-hlec to create the appropriate function and class. Here is what generate-hlec would do for the bundle template:

1) Create a class called bundle-task. Since there is no task superclass specified in the hlec definition, the default superclass of evolution-transformation would be used as the superclass for the bundle-task class.

¹ Note that, in this context, transformation simply refers to the function in the body of an atomic task and includes actions such as the invocation of an editor.

² The Specification Assistant originally referred to evolution transformations as "High-Level Editing Commands" or HLEC. Our original implementation of the task function handled only these evolution transformations (at the time called HLECs). The reference remains "under the hood" of the Concept Demo to this day.

2) Look for a function in the CD package called "bundle". If no such function is found an error is signaled. If the function is found, a new function called bundle-hlec is generated. This new function wraps code around the bundle function to store the arguments of the executing task, save the state of the knowledge-base, update presentations, etc.

To invoke an instance of the bundle-task from a product-based presentation, there must be a context-driven menu option that calls the function bundle-hlec. Calling this function will result in an instance of the bundle-task being created. To invoke an instance of the bundle-task from a process presentation, the task must be generated as a result of intelligent assistance with some or all of its parameters already instantiated. It could then be accessed from a process presentation and sent an execute-task message.

The relevant files for transformation functions are all under the ../concept-demo/main directory. They are all files beginning with the prefix "hlec-" as well as hlec-lisp, create-hlec, create-hlec-lisp, hlec-grammar, hlec-foundations, generate-hlecs-lisp, initialize-hlecs-lisp, and composite-hlecs.

3.5.4 Analyze products

The function update-agenda is called from within each generated hlec function. This function first calls update-newly-resolved-issues which looks at all the existing issues and tests if their conditions for resolution are now satisfied. It sends a make-issue-resolved message to all issues that are now satisfied. Update-agenda then calls analyze-current-state for the current kb-module. Analyze-current-state calls syntactic analysis and resource analysis (these are very similar to the static analysis and resource analysis routines in the Specification Assistant) for that module. Syntactic analysis captures messages generated by the Refine compiler (see sections 4.5.6 and 4.5.7 in the Refine User's Guide) and turns them into appropriate instances of subclasses of the issue class. Resource analysis compares information from re::flow-datum attributes to the legal access specifications (used-by, exports, etc.) on ERSLa KB modules and generates issues for objects that violate the KB module access specifications.

The relevant files for product analysis are all under the ../concept-demo/main directory. They are: static-analysis, resource-analysis, method-op-resource-analysis, kb-states, and create-resource-analysis-issues.

3.5.5 Analyze process

After updating existing issues and creating new issues, update-agenda calls create-tasks-for-module which looks at all new issues and attempts to create possible resolution tasks.

The relevant files for process analysis are all under the ../concept-demo/main directory. They are basic-cd-task-methods, cd-task-methods, issue-resolution-tasks, and agenda.

3.5.6 Update presentations

The Concept Demo presentation-based interface is discussed in detail in section 5.4.2 of the User's Manual.

3.6. Evolutionary Development of a Product-Level KBSA

As discussed in the introduction to this section, the main design goal for the Concept Demo was to develop an architecture for a product-level tool that could provide dramatic improvements over existing CASE tools using current knowledge-base and CASE technology, while also supporting evolution into a full-fledged KBSA. In this section, we discuss how the Concept Demo architecture meets this goal. We will refer to the product-level tool that could be built using existing knowledge-base and CASE technology as a "Knowledge-Based CASE Tool" or KB-CASE. We will refer to the tool that the knowledge-based CASE tool would eventually evolve into using the traditional term "Knowledge-Based Software Assistant" or KBSA.

3.6.1 A Knowledge-Based CASE Tool

The Concept Demo illustrates that it is possible to build a tool today that would look to users like a traditional CASE tool, while providing many of the benefits of formal specification languages and process-based environments. The "transformations" used by such an environment could consist primarily of tool invocations and low-level specification evolution transformations that replicate changes made using existing tools on representations such as E/R and data flow diagrams. The code generation for such a tool would depend on the non-functional requirements for the system being developed. For single user systems running on powerful machines such as workstations or advanced PCs, code generation could simply consist of compiling the executable specification language. For systems that required standard types of architectures (such as multi-user, client-server applications) meaning-preserving transformations can be developed with those specific architectures as targets. For example, high-level specifications could be annotated to indicate client-processes and server-processes and code could then be generated based on those annotations. Finally, for systems with very hard non-functional requirements (such as custom designed, highly distributed systems) code generation could include both meaning-preserving transformations and hand coding. Even where hand-coding was required, KB-CASE's front-end and general process support would make the KB-CASE tool a dramatic improvement over existing CASE technology.

A KB-CASE tool would provide the following advantages over current CASE tools.

Conceptual integration of graphic representations. In traditional CASE tools, there are no formal semantics for the various representations used, or there is only one primary representation which may have a formal basis (e.g., state charts) but can only describe certain aspects of the system (such as control flow) and makes specification of other aspects (such as constraints on static data) very difficult. In a KB-CASE tool, the graphic interfaces would be a front end to the specification language. The specification language would provide a consistent semantics for each representation and for understanding how one representation relates to another.

Analysis and prototyping. The formal semantics of the underlying specification language would facilitate writing analysis routines, because that specification language would be independent of the graphic interfaces. The executability of the specification language would provide prototyping capabilities early in the life-cycle.

Process traceability, guidance, and feedback. The KB-CASE tool would be able to use the Concept Demo architecture to provide sophisticated process support such as keeping track of the rationale for design steps, describing problems, suggesting solutions, tracking development steps against a process model, automating collection of project management data, and automating routine project-management tasks.

3.6.2 Evolution to a KBSA

There are two types of issues that must be addressed to transfer new technology such as KBSA: social issues and technical issues. The architecture of the Concept Demo addresses both types of issues for the evolution from a KB-CASE tool to a KBSA.

3.6.2.1 Social Issues: Acceptance of New Techniques and Styles of Work

If we currently possessed a product-level KBSA that could demonstrate an order of magnitude improvement in productivity for users trained in formal specification and other KBSA techniques, the transfer of such a KBSA into every day use in the DoD and U.S. industry would still be a difficult task. KBSA requires the use of many new techniques. Some of these techniques — such as iterative development and object-oriented design and programming — are already gaining wide acceptance. However, other techniques — such as formal specification languages and process-based environments — are not currently accepted by most developers. In our experience presenting KBSA concepts to developers, we have encountered skepticism about the value of formal specification languages, vis-a-vis the cost of learning to use them. Similarly,

several experiments with process-based environments show that people often view such environments as intrusive and have very negative reactions to them.

The Concept Demo architecture facilitates the acceptance of formal specification languages and process-based environments by providing many of their advantages, while preserving familiar representations and work styles. This will allow developers to adopt the new KBSA techniques gradually. Understanding will come with use. Ambitious developers will be curious about the underlying KB repository representation and will naturally begin to investigate and use the formal specification language in addition to the familiar graphic representations. Similarly, the benefits of a process-based environment could initially be provided through interfaces that were primarily product-oriented – as are almost all current commercial CASE tools. As developers began to understand and accept the idea of process recording, feedback, and guidance they would gradually adopt process-based representations.

3.6.2.2 Technical Issues

Our experience with KBSA techniques leads us to conclude that while formal languages and transformations can cover many of the types of development activities needed to develop actual systems, there are still gaps in the coverage provided by this approach. We believe that it is not currently possible to field a product that would be completely based on transformations and formal languages and capable of building most types of systems. Thus, a central technical issue to solve in regard to transferring KBSA technology is to develop an architecture that can smoothly integrate conventional and KBSA development representations and techniques. This integration should be done in a modular fashion, so that individual tools that use conventional representations and techniques can incrementally be replaced by KBSA tools in a way that minimizes the effect on the rest of the environment.

As discussed above, the Concept Demo architecture achieves this integration of traditional tools with KBSA tools in one consistent process-based environment. The task structure in the Concept Demo can be used to invoke, record, and suggest the use of manual and transformation-based tools or a combination of the two.

The modular nature of the architecture facilitates incremental replacement of conventional tools by KBSA tools. By separating out interface, analysis, product/process representation, and transformation into distinct knowledge-base modules, it is possible to add or replace components in any of these areas with little or no impact on the rest of the environment. Some of the specific ways that the Concept Demo architecture provides this modularity are:

- The presentation-based interface separates the display and update of the knowledge-base from the transformations. Thus, transformations can

be written and added with virtually no concern for user-interface updating. Similarly, new presentations can be added without altering the transformations.

- The general "Accept Values" and context-driven menus functionality further modularizes the interface by separating out the invocation of tasks from specific presentations. This makes it possible to specify how and when to invoke tasks independently from specific presentations. It also makes it possible to add menus for the invocation of new tasks without modifying the code for existing presentations.
- The separation of the analysis of products and the analysis of process makes it possible to add new analysis routines that can be used by developers, even when the process guidance functionality does not yet exist to generate tasks based on the analysis. This is currently done in the Concept Demo, which utilizes all of the error and warning messages generated by the Refine compiler to generate issues that can be very useful for developers, even though the process analysis functionality is only capable of generating suggested resolution tasks for a few of the most common types of issues.

4 The Formal Validation Extension to the KBSA Concept Demo Project

The goals of the formal validation extension to the KBSA Concept Demo project were to study the techniques used in the KBSA paradigm to improve the software development process, identify development phases where formal validation is necessary, and then explore formal validation techniques that can be applied to those phases. The KBSA Concept Demo is well-suited to the application of formal techniques. Its use of a formal specification language encourages the use of formal techniques for specification validation and verification, and transformation of specifications into efficient code using formal meaning-preserving transformations. The Software Refinery environment that the Concept Demo is built on also allows developers to treat programs as data and to manipulate them formally.

In the KBSA paradigm, the requirements of a software system are acquired, analyzed, and formalized to obtain a formal software specification. The formalization process is an evolutionary one that starts with a high-level preliminary specification and ends with a detailed, formal specification. The formal specification is transformed into an efficient implementation using a transformational program derivation system such as the Development Assistant. Meaning-preserving transformations (MPTs) are used in this process. Once the transformations used are proven to be meaning-preserving, formal validation is not necessary.

Evolution transformations (ETs) are used to formalize informal requirements to produce a formal specification. ETs are meaning-changing transformations that introduce systematic changes into the specification at a syntactic level. However, the ET approach leaves the task of validating the constructed specification to the user. A formal validation component can provide valuable assistance to the user in this task.

After further investigation, we determined that the task of validating large, formal specifications is as difficult as validating programs. However, the use of ETs in the specification development process allows us to incrementally validate the specification as it is being developed. For incremental validation, ETs must produce small, well-defined changes to the meanings of specifications. We therefore analyzed the ETs in the Concept Demo and the Knowledge-Based Specification Assistant to determine if they produced small, meaningful changes. Section 4.1 presents our analysis of some of these evolution transformations. Unfortunately, we found that these ETs work at a syntactic level; they only ensure that the resulting specification is syntactically correct but it might not be semantically meaningful. New ETs were needed to assist in the incremental validation process.

To find such ETs, we conducted a case study in formal specification development. We developed a formal specification of the Aesthetic Graph Layout (AGL) problem (used as part of the Concept Demonstration system and described in an earlier report) paying special attention to the process of formalization. This study is described in Section 4.2. As a result of this study, we realized that the process of formalization can be best supported by reuse: the existence of a repository of validated, reusable components that can be retrieved, adapted, and composed can significantly simplify the process of specification construction and validation. The operations for retrieval, adaptation, and composition are the new ETs in the specification development process. We found that ordinary adaptation and composition operators resulted in specifications whose properties could not be easily determined by their components. The adaptation and composition operations in a reuse environment should precisely identify parts that change as a result of their

application; these operations should be based on a theory of reuse.

We have since started work on the development of a theory of reuse. The theory of reuse will have operations for retrieval, adaptation, and composition of reusable components. Some composition operations resulting from our efforts in this direction are presented in Section 4.3. An executable implementation of these operations is described in Section 4.4. In future, we will continue our work in this direction to explore the operations in the theory of reuse in greater detail and consider the possibility of implementing algorithms for some such operations.

4.1 Analyzing Evolution Transformations

Here we present an analysis of a representative sample of evolution transformations in the Knowledge-Based Specification Assistant and the KBSA Concept Demo to evaluate their suitability for incremental validation. We present the following ETs: absorb class, create attribute inverse, delete KB object, redefine class, and splice communicator.

We assume an algebraic semantics for the specification language [Wirsing90]. We analyze each ET by the way it changes the class of algebraic models of the specification. We describe this change using specification constructor operators discussed by Wirsing [Wirsing90].

4.1.1 Absorb class

It takes an **absorbee** class and an **absorber** class and the **absorber** absorbs the **absorbee**.

Let A be the absorber and B be the absorbee. Let $D_1 \dots D_m$ be the descendents of B . The following actions need to be performed to get a meaningful specification as a result of applying this ET:

- Redefine all the attributes and methods of B for A . This includes all the methods and attributes inherited by B from its ancestors and which are exported by B or any of its descendents.
- Change the subsort relationship of the children of B to make them children of A .
- Delete B from the specification.

Let SP be the complete specification before the ET is applied. The models of the new specification obtained as a result of applying this ET to SP can be related to the models of SP as follows:

1. Extend the models of SP by adding sorts $D_1^0 \dots D_m^0$ as subsorts of A (where each D_i^0 is a copy of the corresponding D_i), adding operations to A corresponding to each attribute and method of B , and adding axioms to A corresponding to the axioms of B . If some class E has a method f that has B or some D_i in its rank, then define a new method f^0 that has A or the corresponding D_i^0 in its rank.
2. Derive the result of step 1 using a specification morphism that forgets B , its descendents, and the old methods in other classes that have been redefined.
3. Restrict the result of step 2 for all sorts except B , $D_1 \dots D_m$ to get rid of the junk elements that might be present in the ancestors of B .
4. Rename all D_i^0 s to D_i and f^0 s to f .

4.1.2 Create attribute inverse

Here we assume that f is a total function.

This ET takes a function f and creates f^1 .

Case	f
1.	injective, surjective
2.	injective, not surjective
3.	not injective, surjective
4.	not injective, not surjective

1. $\text{mod}(\text{create-attribute-inverse}(f, f^1, SP)) = \text{mod}(SP)$ when f is bijective.

When f is bijective the axiom added by create-attribute-inverse is $\forall x f^1(f(x)) = x$

I.e., it does not create any new junk or confusion.

2. $\text{mod}(\text{restrict}(\text{derive}(\text{create-attribute-inverse}(f, f^1, SP)))) = \text{mod}(SP)$

The axiom being added is: $f^1(x) = y \text{ iff } f(y) = x$

Here restrict is w.r.t all sorts of SP , and derive forgets f^1 , i.e., derive is w.r.t. in .

I.e., it may create new junk because f^1 is a partial function.

3. $\text{mod}(\text{derive}(\text{create-attribute-inverse}(f, f^1, SP))) = \text{mod}(SP)$

$$\forall x, yx \in f^1(y) \text{ iff } f(x) = y$$

f^1 returns a set of the domain type of f . Here there are 2 possibilities. The set of domain type of f may already exist, in which case this ET does not create any new junk; it simply identifies all new terms with some sets of the domain type. For this case, derive simply forgets f^1 .

The second case occurs when set of domain type of f does not exist in the original specification. In this case, a new carrier set for this set of domain type of f is created. Here derive simply forgets this new carrier set as well.

4. $\text{mod}(\text{restrict}(\text{derive}(\text{create-attribute-inverse}(f, f^1, SP)))) = \text{mod}(SP)$

Because f^1 is partial, it can generate junk in the set of domain type of f . If the set of domain type of f already existed, restrict will get rid of this new junk. On the other hand, if the set did not exist, derive forgets the whole sort and the junk wont matter anyway.

The axiom added is $\forall x, yx \in f^1(y) \text{ iff } f(x) = y \wedge f(z) \neq y \Rightarrow z \notin f^1(y)$

4.1.3 Delete KB object

Deleting an axiom:

$\text{Mod}(\text{delete}([\Sigma, E], e)) = C$ where e is an axiom being deleted, C is a class of algebras, and $[\Sigma, E]$ is the specification on which delete acts. $C + [\Sigma, e] = [\Sigma, E]$.

We are going to ignore this because the result of deleting sorts or operations symbols without clean-up results in a non-specification. (A specification can only use symbols in the language (signature)).

4.1.4 Redefine class

This moves a class from one point in the hierarchy to another. This class, C , could inherit properties from its old superclass, C_{so} . Also, C could be used or imported by other classes. When C is moved, all its children move with it. All the properties that it inherits from C_{so} that are needed for defining things in C or its children, are redefined.

The meaning changes as follows. First a C' is added at the new location, and wherever C was being imported, C' is also imported. Whenever there is an axiom of the form $\phi(f)$ where f is a function symbol with C in its rank we add an axiom of the form, $\phi'(f')$, which requires adding a new operation symbol f' with C replaced by C' .

Then we delete C using the derive operation and use the restrict operation to get rid of all references to C in all classes importing C originally. Then we rename C' to C , f' to f , and so on.

We repeat this process of replacing C by C' for all descendant classes of C .

4.1.5 Splice communicator

Class A has an attribute f of type F which it does not export. Class B has a function g which needs the value of $f(a)$. So a new attribute h of type C is added to class A . Class C has an attribute k of type f . An axiom is added to class A equating the values of attributes $f(a)$ and attribute $k(h(a))$. A exports h and C exports k . Thus B can now access $f(a)$ through $k(h(a))$.

4.1.6 Conclusion

The ETs in both KBSpecA and the Concept Demo were designed to introduce systematic changes into specifications. They provided high-level editorial and book-keeping support to ensure that the resulting specifications were syntactically correct, that is, the specifications could be parsed correctly by a specification language parser. However, the ETs were not designed to produce semantically meaningful specifications, that is, the resulting specifications could be inconsistent or incomplete. In addition, the ETs were not intended to be used for incremental validation, and hence were not designed to produce small, identifiable changes in the meaning of the specification. Since a small syntactic change can change the meaning of a specification significantly, application of ETs can produce large semantic changes in the meanings of specifications. Therefore there is a need for designing new ETs that can produce such small, identifiable changes for assistance in evolutionary development with incremental validation.

4.2 An Algebraic Specification of an Aesthetic Graph Layout Problem

The industrial approach to developing software is to go directly from informal requirements to code. Code is then validated against requirements by proving that it meets the requirements. But code contains optimizations that spread information and implement simple abstractions with complex realizations [?]. These optimizations hinder understanding and analysis of code, making validation extremely hard. Any errors that are detected during validation cause extensive rework. Most seriously, the difficulty of the validation task significantly increases the likelihood of overlooking errors in the code. These overlooked errors can potentially be hazardous in critical applications.

Validation can be made easier by first developing a formal specification from the requirements and then validating the specification against requirements [?, ?]. Validating specifications is easier

than validating code because specifications contain no optimizations or implementation details, making specifications easier to understand and analyze. As many studies have shown [?], the amount of work required to correct software errors is a function of the phase in which the corrections are made. Since specification construction occurs very early in the software development process, the amount of work required to correct errors in the specification is relatively small. Also, the likelihood of finding errors or inconsistencies is increased because understanding specifications is easier than understanding code.

By introducing specification construction and validation into the software development process, we can divide the process into two parts:

1. Construction of validated formal specifications from requirements.
2. Derivation of correct implementations from validated specifications.

We have used this process to construct a formal specification for an aesthetic graph layout (AGL) problem and to derive an implementation from this formal specification. Here, we focus our attention on the specification construction process. The implementation for graph layout derived from the formal specification is described elsewhere [?]. An extended version of the specification described here is being used to develop an industrial strength implementation of an aesthetic graph layout system that will be part of a commercial product.

The rest of this paper is organized as follows. In Section 4.2.1, we describe the informal requirements for the aesthetic graph layout problem. We describe the development of the formal specification itself in Section 4.2.2. We emphasize the *process* of formalization as opposed to the constructed formal specification. We want the specification to have the following properties: correctness, completeness, consistency, abstractness, readability, and executability. In Section 4.2.3, we analyze the specification construction process for each property mentioned above and support our arguments with examples from the case study. We describe the difficulties encountered during the formalization process and conclude by suggesting technological advances that can help alleviate such difficulties. Appendix ?? contains the complete text of our specification.

4.2.1 The Aesthetic Graph Layout Problem

Graphs are ubiquitous as a means of representing information and structure. Their usage has increased dramatically with the advent of graphic tools. An extensive survey of graph layout algorithms and tools can be found in [?].

Most approaches to graph layout use one of two modes: *static* or *dynamic*. In static graph layout, a combinatorial description of the graph is used to produce a layout for the entire graph in a single step. In dynamic graph layout, the graph is constructed incrementally; nodes and edges are added one at a time, and the system redraws the graph after each step to satisfy the aesthetic criteria. The dynamic mode of graph layout is clearly more appropriate for interactive construction of graphs. However, with the dynamic graph layout approach, the topology of the graph layout can change drastically even for minor changes in the graph. These drastic changes in topology can make it very hard to follow the semantic evolution of the graph.

In this paper, we discuss a formal specification of a software system that supports an *incremental* mode of graph layout. In incremental graph layout, nodes and edges are added to the graph one at a time. The topology of the existing graph layout is preserved in the new embedding, except for the possible introduction of some new edge crossings. The incremental mode of graph

layout overcomes the drawbacks of dynamic graph layout by never changing the graph drastically. The system never repositions nodes; however, there may be edge crossings even for planar graphs.

In the software engineering domain, dataflow diagrams and entity-relationship diagrams are two of the most widely used representation techniques. Traditionally, these diagrams have been drawn using the *orthogonal graphic standard* [?]; every node is a rectangle and every edge is comprised of horizontal and vertical segments.

In this paper, we specify a system for incremental graph layout using the orthogonal graphic standard. We use the following primary aesthetic criteria:

- minimize the number of crossings among edges,
- minimize the number of bends in each edge, and
- minimize the length of each edge.

In addition, a node cannot overlap another node, an edge cannot cross a node, and an edge cannot overlap another edge.

4.2.2 Development of the Specification

We express our specification in a formal language based on first order logic. Using this language, we can describe each abstract data type as a separate module.

We use the following criteria to guide the development of the specification: correctness, completeness, consistency, abstractness, and readability. To build the AGL specification, we define a top-level abstract data type called AGL with operations such as **add-node**, **add-edge**, **delete-node**, and **delete-edge**. These operations in turn require definition of data types for node, edge, line segments, sequence of line segments, and point.

In this section, we first describe the specification language and then define each abstract data type needed for the AGL specification.

4.2.2.1 Specification Language and its Semantics We are interested in describing the objects in the AGL domain, and the operations that we can perform on the objects. Objects can be represented as elements of sets and operations can be considered to be functions on these elements. A collection of sets and functions together form an algebra. Hence a specification language with algebraic semantics is most suitable for modeling the AGL problem.

A specification in our specification language is a collection of specification components. Each specification component introduces one new sort and defines operations on this sort and the properties of these operations. Each specification component SP consists of a new sort, a set of sorts that are defined elsewhere, a set of operations on the sorts, and a set of axioms (expressed in first-order logic) [?, ?]. The set of sorts and the set of operations are together called the *signature* of the specification. Intuitively, each specification component corresponds to the definition of one abstract data type.

Definition 1 *Specification Component.* A specification component SP consists of a pair $\langle \Sigma, E \rangle$ where

- $\Sigma = \langle S, B, F \rangle$ is the *signature* of SP . S is the new *sort*, B is the set of sorts used by SP that are defined in other specification components, and F is the set of *operations*;

- E is the set of *axioms* of SP . ■

For a simple example of a specification component, see the specification of POINT in Appendix ??.

The specification language has an algebraic semantics. An algebra is a collection of sets and a collection of functions on these sets. An algebraic model of a specification is defined as follows:

Definition 2 Algebraic Model. Let SP be a specification consisting of set of specification components $SP_i = \langle \Sigma_i, E_i \rangle, i \in I$ where $\Sigma_i = \langle S_i, B_i, F_i \rangle$. Let $S = \bigcup_{i \in I} (\{S_i\} \cup B_i)$ be the sorts of SP , $F = \bigcup_{i \in I} F_i$ be the operations of SP , and $E = \bigcup_{i \in I} E_i$ be the axioms of SP . Then a *model* of SP consists of an *algebra* $A = \langle A_S, F_A \rangle$ where

- $A_S = \{A_s \mid s \in S\}$ is a collection of sets called the *carrier sets* of the algebra;
- $F_A = \{f_A \mid f \in F\}$ is a collection of *functions* of the algebra, such that if the rank of f is $s_1, \dots, s_n \rightarrow s$, then f_A is a function from $A_{s_1} \times \dots \times A_{s_n}$ to A_s ;
- A satisfies all the axioms in E . ■

For a more detailed description of algebraic semantics, including the meaning of satisfiability of axioms, see Wirsing's paper [?] or Srinivas's report [?].

Given a specification in this specification language there is a class of algebraic models for the specification. If the specification satisfies certain conditions [?], there exists a distinguished algebra (up to isomorphism) in the class of algebraic models. This distinguished algebra, called the *initial algebra*, has a unique homomorphism to each of the other algebraic models of the specification. We adopted the initial algebra semantics for our specification language.

4.2.2.2 The AGL Specification Component We use an object-oriented approach to develop the AGL specification. We use existing standard specification components for basic types such as natural numbers, real numbers, integers, Booleans, sets, and sequences. We then create a specification component to introduce a new sort called AGL. Conceptually, the sort AGL represents the displayed graph. We define the operations *AGL-edges* and *AGL-nodes* to characterize the edges and the nodes of the AGL, respectively. Each node of an AGL is rectangular. Each edge of an AGL is a sequence of horizontal and vertical connected line segments. We define operations for adding and deleting edges and nodes to the AGL, and for creating a trivial AGL with no nodes or edges. The AGL specification component has the following operations at this point:

```

AGL-create:  $\rightarrow$  AGL
AGL-add-node: POINT  $\times$  REAL  $\times$  REAL  $\times$  AGL  $\rightarrow$  AGL
AGL-delete-node: NODE  $\times$  AGL  $\rightarrow$  AGL
AGL-add-edge: NODE  $\times$  NODE  $\times$  AGL  $\rightarrow$  AGL
AGL-delete-edge: EDGE  $\times$  AGL  $\rightarrow$  AGL
AGL-nodes: AGL  $\rightarrow$  NODE-SET
AGL-edges: AGL  $\rightarrow$  EDGE-SET

```

AGL-create creates a trivial AGL. Note that *AGL-add-node* takes a point, and two numbers representing the dimensions of the node in addition to the AGL itself as arguments, whereas *AGL-delete-node* takes a node and the AGL as arguments. *NODE-SET* and *EDGE-SET* are sets of nodes and edges, respectively.

Most of the axioms associated with these operations are quite simple. The nodes and edges of an empty AGL are empty sets.

$AGL-nodes(AGL-create) = \emptyset$

$AGL\text{-}edges(AGL\text{-}create) = \emptyset$

A node can only be added to an AGL if it does not overlap another node. If this condition is satisfied, then the nodes of the new AGL are all the previous nodes and the new node.

```

AGL-nodes(AGL-add-node(p, x, y, g)) =
  if  $\exists n_1: \text{NODE}, n_1 \in AGL\text{-}nodes(g) \wedge \text{NODE-node-overlap}(n, n_1)$ 
  then AGL-nodes(g)
  else AGL-nodes(g)  $\cup \{n\}$ 
  where  $n: \text{NODE} = \text{NODE-create}(p, x, y)$ 

```

A new node may not overlap any other nodes, but it may overlap existing AGL edges. To add a node to the graph when it overlaps existing edges of the graph, we first remove the overlapped edges from the graph and then add the new node. We then add the deleted edges back to the graph. This requires the addition of some new operations as shown below:

```

AGL-edges(AGL-add-node(p, x, y, g)) =
  if  $\exists n_1: \text{NODE}, n_1 \in AGL\text{-}nodes(g) \wedge \text{NODE-node-overlap}(n, n_1)$ 
  then AGL-edges(g)
  else AGL-edges(AGL-add-edge-set
    (s, AGL-add-non-overlapping-node(p, x, y, AGL-delete-edge-set(s, g)))
    where
      s: EDGE-SET =  $\{e: \text{EDGE} \mid e \in AGL\text{-}edges(g) \wedge \text{NODE-edge-intersect}(n, e)\}$ 
      n: NODE = NODE-create(p, x, y)
  AGL-nodes(AGL-add-non-overlapping-node(p, x, y, g)) =
    AGL-nodes(g)  $\cup \{\text{NODE-create}(p, x, y)\}$ 
  AGL-edges(AGL-add-non-overlapping-node(p, x, y, g)) = AGL-edges(g)
  AGL-add-edge-set( $\emptyset$ , g) = g
  AGL-add-edge-set( $\{e\} \cup s$ , g) =
    AGL-add-edge-set(s, AGL-add-edge(EDGE-source(e), EDGE-target(e), g))
  AGL-delete-edge-set( $\emptyset$ , g) = g
  AGL-delete-edge-set( $\{e\} \cup s$ , g) = AGL-delete-edge-set(s, AGL-delete-edge(e, g))

```

The operation *AGL-add-non-overlapping-node* adds a new node to the graph such that the node does not overlap any existing edges. The operations *AGL-add-edge-set* and *AGL-delete-edge-set* respectively add and delete a set of edges to the graph.

Deleting a node from the AGL has the effect of removing it from the set of nodes and removing all the edges that have that node as either the target or the source.

```

AGL-nodes(AGL-delete-node(n, g)) = AGL-nodes(g) -  $\{n\}$ 
AGL-edges(AGL-delete-node(n, g)) =
   $\{e: \text{EDGE} \mid e \in AGL\text{-}edges(g) \wedge \text{EDGE-source}(e) \neq n \wedge \text{EDGE-target}(e) \neq n\}$ 

```

Deleting an edge only removes the edge from the set of edges of the AGL.

```

AGL-nodes(AGL-delete-edge(e, g)) = AGL-nodes(g)
AGL-edges(AGL-delete-edge(e, g)) = AGL-edges(g) -  $\{e\}$ 

```

To add a new edge to the graph, we specify its source node and its target node. The operation *AGL-add-edge* does not change the nodes of the AGL.

```

AGL-nodes(AGL-add-edge( $n_1, n_2$ , g)) = AGL-nodes(g)
AGL-edges(AGL-add-edge( $n_1, n_2$ , g)) = AGL-edges(g)  $\cup \{\text{EDGE-create}(n_1, n_2, g)\}$ 

```

The complete specification of the AGL component is shown in Appendix ??.

4.2.2.3 NODE Since a graph consists of nodes and edges, we need an abstract data type for graph nodes. In our AGL specification, we assume that each node is rectangular. We specify a node by the location of its center and its extent from the center along the X and the Y axes. We therefore need an operation to create a node given a point and two real numbers. We also

need operations to get the location and size of a node, its corners ¹, and its boundaries. The boundaries of a node is the set of the four lines from one corner of the node to another.

```

NODE-create: POINT  $\times$  REAL  $\times$  REAL  $\rightarrow$  NODE
NODE-position: NODE  $\rightarrow$  POINT
NODE-x-extent: NODE  $\rightarrow$  REAL
NODE-y-extent: NODE  $\rightarrow$  REAL
NODE-top-left-corner: NODE  $\rightarrow$  POINT
NODE-top-right-corner: NODE  $\rightarrow$  POINT
NODE-bottom-left-corner: NODE  $\rightarrow$  POINT
NODE-bottom-right-corner: NODE  $\rightarrow$  POINT
NODE-boundaries: NODE  $\rightarrow$  LINE-SET
NODE-position(NODE-create(p, x, y)) = p
NODE-x-extent(NODE-create(p, x, y)) = x
NODE-y-extent(NODE-create(p, x, y)) = y
NODE-top-left-corner(n) =
  point-create(point-x-coordinate(NODE-position(n)) - NODE-x-extent(n)/2,
    point-y-coordinate(NODE-position(n)) + NODE-y-extent(n)/2)
NODE-boundaries(n) =
  {LINE-create(p1, p2), LINE-create(p2, p3),
    LINE-create(p3, p4), LINE-create(p4, p1)}
  where
    p1 = NODE-top-left-corner(n)
    p2 = NODE-top-right-corner(n)
    p3 = NODE-bottom-right-corner(n)
    p4 = NODE-bottom-left-corner(n)

```

We can add a new node to an AGL only if it does not overlap any other nodes. Therefore we need an operation to determine if two given nodes overlap. Two nodes overlap if their boundaries intersect.

```

NODE-node-overlap: NODE  $\times$  NODE  $\rightarrow$  BOOLEAN
NODE-node-overlap(n1, n2) =
  LINE-SEQUENCE-intersect(set-to-sequence(NODE-boundaries(n1)),
    set-to-sequence(NODE-boundaries(n2)))

```

In the definition above, the operation *LINE-SEQUENCE-intersect* takes two sequences of line segments and determines if they have a point in common. *set-to-sequence* is an operation used to convert a *set* into a *sequence*.

When we add a new node to an AGL, all the existing edges of the AGL that either intersect or overlap the AGL must be redrawn. Therefore we need operations to determine if a given edge intersects or overlaps a node. A node and an edge intersect if they have at least one point in common. A node and an edge overlap if they have at least two points in common. In AGL, node and edge overlap is never allowed, whereas an edge is allowed to intersect a node if the node is either the source or the target of the edge.

```

NODE-edge-intersect: NODE  $\times$  EDGE  $\rightarrow$  BOOLEAN
NODE-edge-overlap: NODE  $\times$  EDGE  $\rightarrow$  BOOLEAN
NODE-edge-intersect(n, e) =
  LINE-SEQUENCE-intersect(set-to-sequence(NODE-boundaries(n)), EDGE-segments(e))
NODE-edge-overlap(n, e) =
  LINE-SEQUENCE-overlap(set-to-sequence(NODE-boundaries(n)), EDGE-segments(e))

```

Finally, to determine if a given node is a source or a target of a particular edge, we want an operation to determine if a given point lies on the boundaries of the node.

```

NODE-has-point-on-boundary: NODE  $\times$  POINT  $\rightarrow$  BOOLEAN
NODE-has-point-on-boundary(n, p) =
   $\exists$  l: LINE, l  $\in$  NODE-boundaries(n)  $\wedge$  LINE-contains(l, p)

```

The entire specification for nodes is given in Appendix ??.

¹The names *NODE-top-left-corner*, *NODE-top-right-corner*, etc. are used to improve readability. In reality if, for example, both the node extents are negative, then *NODE-top-left-corner* will be the bottom right corner of the node.

4.2.2.4 EDGE Here we define the abstract data type for edges of a graph. While we can add a new node to the AGL only if it does not overlap any existing nodes, a new edge can be added to the AGL at any time as long as the new edge's source and target nodes exist in the AGL. The line segments that will form the new edge must not intersect or overlap any other existing nodes, and must also satisfy our other aesthetic criteria as mentioned in Section 4.2.1. Therefore, to create a new edge, we need to specify its source and target nodes as well the existing AGL.

EDGE-create: $\text{NODE} \times \text{NODE} \times \text{AGL} \rightarrow \text{EDGE}$

Given an edge, we need operations to get its source, target, and its line segments.

EDGE-source: $\text{EDGE} \rightarrow \text{NODE}$
 EDGE-target: $\text{EDGE} \rightarrow \text{NODE}$
 EDGE-segments: $\text{EDGE} \rightarrow \text{LINE-SEQUENCE}$
 $\text{EDGE-source}(\text{EDGE-create}(n_1, n_2, g)) = n_1$
 $\text{EDGE-target}(\text{EDGE-create}(n_1, n_2, g)) = n_2$

Creating line segments during edge routing is the most important part of the whole specification. The axioms for *EDGE-segments* specify the properties of all the edges in AGL. Edges are composed of a sequence of vertical or horizontal line segments. These line segments form a connected sequence (that is, the end point of each line segment is the beginning point of the next line segment in the sequence). The line segments should connect the source and target nodes of the edge, and they should not intersect with any other node (that is, edges should not have any points in common with nodes other than their source and target nodes). The line segments should not overlap (that is, have more than one point in common with) the source and target nodes, and they should not overlap the line segments of any other existing edges of the AGL. Any sequence of line segments that satisfies these requirements is a candidate for an edge of the graph. From the set of edge candidates, we choose one of the sequences which best satisfies our aesthetic criteria to be the actual line segments forming an edge in the AGL.

$\text{AGL-most-aesthetic-edge-segments}(g, n_1, n_2, \text{EDGE-segments}(\text{EDGE-create}(n_1, n_2, g))) = \text{true}$
 $\text{AGL-most-aesthetic-edge-segments}(g, n_1, n_2, s) =$
 $\text{AGL-potential-edge-segments}(g, n_1, n_2, s)$
 $\wedge \forall s_1 : \text{LINE-SEQUENCE}, \text{AGL-potential-edge-segments}(g, n_1, n_2, s_1)$
 $\Rightarrow (\text{AGL-edge-segments-cost}(g, s) \leq \text{AGL-edge-segments-cost}(g, s_1))$
 $\text{AGL-potential-edge-segments}(g, n_1, n_2, s) =$
 $(\forall l : \text{LINE}, l \in s \Rightarrow (\text{LINE-horizontal}(l) \vee \text{LINE-vertical}(l)))$
 $\wedge \text{LINE-SEQUENCE-connected}(s) \wedge \text{NODE-connects}(n_1, n_2, s)$
 $\wedge \text{AGL-no-other-node-intersections}(g, n_1, n_2, s)$
 $\wedge \neg \text{LINE-SEQUENCE-overlap}(s, \text{set-to-sequence}(\text{NODE-boundaries}(n_1)))$
 $\wedge \neg \text{LINE-SEQUENCE-overlap}(s, \text{set-to-sequence}(\text{NODE-boundaries}(n_2)))$
 $\wedge \forall e : \text{EDGE}, e \in \text{AGL-edges}(g) \Rightarrow \neg \text{LINE-SEQUENCE-overlap}(s, \text{EDGE-segments}(e))$
 $\text{NODE-connects}(n_1, n_2, s) =$
 $\text{NODE-has-point-on-boundary}(n_1, \text{LINE-from}(\text{first}(s)))$
 $\wedge \text{NODE-has-point-on-boundary}(n_2, \text{LINE-to}(\text{last}(s)))$
 $\text{AGL-no-other-node-intersections}(g, n_1, n_2, s) =$
 $\forall n : \text{NODE}, (n \in \text{AGL-nodes}(g) \wedge n \neq n_1 \wedge n \neq n_2)$
 $\Rightarrow \neg \text{LINE-SEQUENCE-intersect}(s, \text{set-to-sequence}(\text{NODE-boundaries}(n)))$
 $\text{AGL-edge-segments-cost}(g, s) =$
 $((\sum_{l \in s} \text{LINE-length}(l)) * \text{AGL-edge-length-cost-factor}())$
 $+ ((|s| - 1) * \text{AGL-edge-bends-cost-factor}())$
 $+ ((\sum_{e \in \text{AGL-edges}(g)} \text{LINE-SEQUENCE-intersections}(s, \text{EDGE-segments}(e)))$
 $* \text{AGL-edge-crossings-cost-factor}())$

We use *AGL-edge-length-cost-factor*, *AGL-edge-bends-cost-factor*, and *AGL-edge-crossings-cost-factor* as the constants used to assign weights to each aesthetic criterion used for AGL edge layout.

A complete specification of the EDGE specification component is shown in Appendix ??.

Note that some of the operations described above are moved to other specification components based on their signatures. For example, the operation *NODE-connects* is moved to the NODE component since the axiom for *NODE-connects* uses operations defined for NODE and does not use any operations defined for EDGE.

4.2.2.5 LINE Edges in AGL are composed of line segments. Line segments also form the boundaries of nodes. We specify line segments by their endpoints. In AGL, line segments are directed. Also, we only consider vertical or horizontal line segments. We therefore need operations to determine the orientation of line segments.

```

LINE-create: POINT  $\times$  POINT  $\rightarrow$  LINE
LINE-from: LINE  $\rightarrow$  POINT
LINE-to: LINE  $\rightarrow$  POINT
LINE-horizontal: LINE  $\rightarrow$  BOOLEAN
LINE-vertical: LINE  $\rightarrow$  BOOLEAN
LINE-from(LINE-create( $p_1, p_2$ )) =  $p_1$ 
LINE-to(LINE-create( $p_1, p_2$ )) =  $p_2$ 
LINE-horizontal( $l$ ) = (POINT-Y-coordinate(LINE-from( $l$ )) = POINT-Y-coordinate(LINE-to( $l$ )))
LINE-vertical( $l$ ) = (POINT-X-coordinate(LINE-from( $l$ )) = POINT-X-coordinate(LINE-to( $l$ )))

```

Since the length of line segments that form an edge is one of our aesthetic criteria, we need an operation to calculate the length of a line segment. We also need operations to determine if two line segments have one or more points in common because such operations will be used to determine if an edge intersects or overlaps another edge or node.

```

LINE-length: LINE  $\rightarrow$  REAL
LINE-intersect: LINE  $\times$  LINE  $\rightarrow$  BOOLEAN
LINE-overlap: LINE  $\times$  LINE  $\rightarrow$  BOOLEAN
LINE-length( $l$ ) = POINT-distance(LINE-from( $l$ ), LINE-to( $l$ ))
LINE-intersect( $l_1, l_2$ ) =  $\exists p$ : POINT, LINE-contains( $l_1, p$ )  $\wedge$  LINE-contains( $l_2, p$ )
LINE-overlap( $l_1, l_2$ ) =
   $\exists p_1, p_2$ : POINT,  $p_1 \neq p_2 \wedge$  LINE-contains( $l_1, p_1$ )
     $\wedge$  LINE-contains( $l_1, p_2$ )  $\wedge$  LINE-contains( $l_2, p_1$ )  $\wedge$  LINE-contains( $l_2, p_2$ )

```

We determine if a given point lies on a given line segment by substituting the coordinates of the point in the line equation, and by verifying that the point lies in the bounding rectangle of the line.

```

LINE-contains( $l, p$ ) =
  (( $(y - y_1) * (x_2 - x_1) = (x - x_1) * (y_2 - y_1)$ ))  $\wedge$  ( $x_1 \leq x \leq x_2$ )  $\wedge$  ( $y_1 \leq y \leq y_2$ )
  where
     $x$  = POINT-X-coordinate( $p$ )
     $y$  = POINT-Y-coordinate( $p$ )
     $x_1$  = POINT-X-coordinate(LINE-from( $l$ ))
     $y_1$  = POINT-Y-coordinate(LINE-from( $l$ ))
     $x_2$  = POINT-X-coordinate(LINE-to( $l$ ))
     $y_2$  = POINT-Y-coordinate(LINE-to( $l$ ))

```

The complete specification for *LINE* is shown in Appendix ??.

4.2.2.6 LINE-SEQUENCE We use a sequence of line segments to form an edge in AGL. We define all the standard sequence operations such as *first*, *rest*, *last*, *insert*, *delete*, and *find* for the sort LINE-SEQUENCE. We also define some other operations for line sequences that are used by other specification components of AGL. To determine if two edges intersect or overlap, we need operations to determine if two line sequences overlap or intersect. We also need operations to determine the number of times two line sequences intersect.

```

LINE-SEQUENCE-overlap: LINE-SEQUENCE  $\times$  LINE-SEQUENCE  $\rightarrow$  BOOLEAN
LINE-SEQUENCE-intersect: LINE-SEQUENCE  $\times$  LINE-SEQUENCE  $\rightarrow$  BOOLEAN
LINE-SEQUENCE-intersections: LINE-SEQUENCE  $\times$  LINE-SEQUENCE  $\rightarrow$  NAT
LINE-SEQUENCE-overlap( $s_1, s_2$ ) =

```


$$\begin{aligned}
& \exists l_1, l_2: \text{LINE}, l_1 \in s_1 \wedge l_2 \in s_2 \wedge \text{LINE-overlap}(l_1, l_2) \\
& \text{LINE-SEQUENCE-intersect}(s_1, s_2) = \\
& \exists l_1, l_2: \text{LINE}, l_1 \in s_1 \wedge l_2 \in s_2 \wedge \text{LINE-intersect}(s_1, s_2) \\
& \text{LINE-SEQUENCE-intersections}(s_1, s_2) = \\
& \sum_{l_1 \in s_1} \sum_{l_2 \in s_2} \text{if } \text{LINE-intersect}(l_1, l_2) \text{ then } 1 \text{ else } 0
\end{aligned}$$

To form an edge, the segments in a line sequence must be joined to each other. We therefore need an operation to determine if a line sequence is connected.

```

LINE-SEQUENCE-connected: LINE-SEQUENCE → BOOLEAN
LINE-SEQUENCE-connected([]) = TRUE
LINE-SEQUENCE-connected([l]) = true
LINE-SEQUENCE-connected(prepend(l1, prepend(l2, s))) =
  LINE-to(l1) = LINE-from(l2) ∧ LINE-SEQUENCE-connected(prepend(l2, s))

```

The complete specification for *LINE-SEQUENCE* (except for the standard sequence operations) is shown in Appendix ??.

4.2.3 Discussion of Issues

Balzer [?, ?] identifies the following as sources of informality in software specifications:

- *Abbreviation.* Abbreviation is the use of domain-specific vocabulary to refer to complex pieces of knowledge in that domain. We view abbreviation as simply a form of reuse. Since we did not use a repository of domain-specific reusable components, we do not discuss this issue further in this section.
- *Ambiguity and Inaccuracy.* Ambiguity in specifications refers to multiple interpretations of specifications. A specification is inaccurate if it does not reflect the intent of the user. As both these issues result in incorrect specifications, we treat them as a single issue in Section 4.2.3.1.
- *Inconsistency.* Inconsistency means that the semantics of different operations, specified by the axioms, are contradictory [?]. We discuss consistency in more detail in Section 4.2.3.3.
- *Incompleteness.* A specification is incomplete when descriptions of certain aspects of the software system are missing from the specification. We deal with this issue in Section 4.2.3.2.
- *Poor Ordering.* The requirements may be hard to understand if they are not presented in a proper order. As discussed in Section ??, poor ordering is not an issue if the specification is developed using an object-oriented approach.

While Balzer was interested in the specifications themselves, we are concerned with the *process* of formalizing specifications.

In addition to dealing with the above sources of informality, we wanted the specification to be highly readable and abstract. Since we wanted to use our specification as a basis for deriving an actual implementation of the AGL system, we wanted a specification that could be easily understood by software developers with little training in algebraic specifications. We also wanted our specification to be highly abstract to provide maximum freedom to the developers of the system in choosing among implementation alternatives. Executability of the specification was a secondary goal that we wanted to achieve if it did not compromise the primary goals.

We discuss how our specification development process was guided by the above goals. We also describe situations where the goals conflicted and we had to make trade-offs in achieving the goals.

4.2.3.1 Correctness One of the most important steps in building a specification is its validation with respect to requirements. During specification validation, errors are detected and eliminated. This is a difficult problem because requirements are non-formal and hence cannot be directly used to formally validate the specification.

The validation procedure used in our AGL case study can be best described as "validation by inspection." We analyzed the axioms for each operation to determine if the operation met the specified requirements. During this analysis, we identified three principal sources of errors in the AGL specification.

- **Ambiguous and Incomplete Requirements**

The original set of requirements for the AGL system stated that while edges could not intersect nodes, they could intersect other edges. While analyzing the axioms for edge routing, we discovered that if we allowed edges to intersect with no other constraints, two edges could have overlapping line segments. After further deliberation, we decided that overlapping edges should not be allowed and added further constraints to the AGL specification to accomplish that. We feel that detecting ambiguities of this nature early in the software development process is a principal benefit of developing formal specifications.

- **Interdependencies Between Requirements**

The AGL system has the requirement that an edge should not pass through the source or the target node. The AGL system also needs to meet the aesthetic criterion of minimizing the lengths of edges. An early version of the AGL specification relied on achieving the former requirement as a side-effect of satisfying the latter aesthetic criterion – if an edge passes through the source or the target nodes, there is always another edge with a shorter length that does not pass through the node(s). After further analysis, we realized that this reliance on interdependencies between requirements to meet the requirements affected readability as well as maintainability. For instance if minimizing the length of edges ceases to be an aesthetic criterion at some point in the life of the software system, the removal of this criterion from the specification would result in a specification that no longer meets one of its requirements. Therefore we represented the requirement that edges should not pass through source or target nodes explicitly in the specification.

- **Changes to the Specification**

The axioms that describe the properties of the operation *AGL-add-node* state that if the new node being added overlaps any existing edges, the overlapped edges should be removed and rerouted after adding the node. In an early version of the specification, we discovered that the edges were being removed and rerouted, but the addition of the new node before rerouting the edges had been overlooked.

In correcting this error, we introduced another error into the specification. The erroneous new axiom was as follows:

```

AGL-edges(AGL-add-node(p, x, y, g)) =
  if  $\exists n_1: \text{NODE } n_1 \in \text{AGL-nodes}(g) \wedge \text{NODE-node-overlap}(n, n_1)$ 
  then AGL-edges(g)
  else AGL-edges(AGL-add-edge-set
    (s, AGL-add-node(p, x, y, AGL-delete-edge-set(s, g))))
  where
    s: EDGE-SET = {e:EDGE | e  $\in$  AGL-edges(g)  $\wedge$  NODE-edge-intersect(n, e)}
    n: NODE = NODE-create(p, x, y)

```

```

POINT-create: REAL  $\times$  REAL  $\rightarrow$  POINT
POINT-X-coordinate: POINT  $\rightarrow$  REAL
POINT-Y-coordinate: POINT  $\rightarrow$  REAL
POINT-X-coordinate(POINT-create(a,b)) = a
POINT-Y-coordinate(POINT-create(a,b)) = b

```

Figure 1: Sufficient Completeness

The use of *AGL-add-node* in the definition results in a non-terminating sequence of deletion of an empty set of edges from the graph. We fixed this error by replacing the use of *AGL-add-node* in the definition by a new operation called *AGL-add-non-overlapping-node*.

4.2.3.2 Completeness We used the notion of *sufficient completeness* (defined below) to ensure that our specification is complete. As mentioned earlier, each specification component in the AGL specification corresponds to the definition of one abstract data type (ADT). We divide the set of operations defined on an ADT into two sets, *S* and *O*. The set *S* contains the operations whose range is the ADT being specified. The set *O* contains operations that map values of the type ADT onto other types; these are often called *behavior operations*. A data type specification is sufficiently complete if there exists a subset of its non-behavior operations, called *constructors*, such that any expression returning the abstract data type can be reduced, by using the axioms for substitution, to an expression consisting only of constructors [?, ?]. Figure 4.2.3.2 shows a fraction of the *POINT* specification. This specification is sufficiently complete because *POINT-create* is the only constructor and expressions containing *POINT-X-coordinate* and *POINT-Y-coordinate* can be reduced to real numbers (here, we make the assumption that our reusable specification component for real numbers is sufficiently complete).

The general problem of determining whether a specification is sufficiently complete is undecidable [?]. To ensure the sufficient completeness of our specification, we identified the sets of constructors very early in the specification development process, and defined axioms such that the most general forms of all the expressions using all non-constructor operations could be reduced to expressions consisting only of constructors. This is a general method useful for creating sufficiently complete specifications and eliminates the need to analyze the final specification for sufficient completeness. This method can be used more profitably if the specification is modular, with well-defined dependencies between the components of the specification.

There can be other forms of incompleteness in specifications. Wing, in her study of twelve specifications of the library problem [?], identifies six major incompleteness categories:

- **Initialization.** A specification should state the properties of the initial state of the system. Our algebraic specification does not have a notion of states, but the properties of the operation *AGL-create* describe the properties of the system before any nodes or edges are added to the graph.
- **Missing Operations.** A statement of requirements may be incomplete because of its failure to identify all the useful operations that the system could perform. This incompleteness may be carried over into the specification. We identified a few potentially useful operations such as an operation to move a node to a new location or an operation to change the size of a node that are missing from our specification. These operations may be added to the specification in future.

- **Error Handling.** A specification should specify what happens if an error is encountered. This is most useful, for example, in the specification of a user interface. Since we do not specify a user interface in our specification, the only possible errors in our specification are due to some operations being defined as total operations when they should actually be partial operations. We could not find any such errors in our specification.
- **Missing Constraints.** A statement of requirements may fail to identify all the constraints to be satisfied by the objects in the system and their operations. This incompleteness may be carried over into the specification. One such missing constraint in an earlier version of our specification was the missing requirement that edges should not overlap. This constraint has been added to the specification presented in Appendix ??.
- **Change of State.** A specification may be incomplete if it fails to specify what objects change from state to state. In an algebraic specification with no notion of states, this type of incompleteness corresponds to absence of sufficient completeness.
- **Nonfunctional Behavior.** A specification may also be incomplete if it does not address some issues such as human interaction, system constraints, or liveness. We did not consider any non-functional issues such as human interaction or system constraints such as the size of the graph or the display screen in our specification in order to give maximum flexibility to the implementors.

4.2.3.3 Consistency Consistency means that the semantics of different operations, specified by the axioms, are not contradictory. Determining consistency is theoretically an undecidable problem [?]. We ensured the consistency of our specification by making sure at each step in the specification development process that a new axiom was consistent with all the previous axioms. The modularity of specifications is a tremendous asset in analysis of this nature because it limits the size of the sets of axioms that have to be checked for consistency. We feel that tools can be developed to assist specification developers in constructing consistent specifications. We have already initiated work in this direction [?].

4.2.3.4 Readability versus Executability As mentioned earlier, a main goal of our experiment was to create a readable formal specification for an aesthetic graph layout system that could be used by people unfamiliar with the underlying formalism to create an efficient implementation of the system. We believe that we were successful in creating a specification that can be understood fairly easily by those with some knowledge of logic. We had a secondary objective to create an executable specification. To meet this objective, we experimented with converting our algebraic specification into an executable specification in a wide-spectrum language called Refine [?]. During the course of the experiments, we found that in order to achieve executability, we had to discard the full expressive power of first order logic. A typical illustration of the kind of problems we encountered is the definition for the operation *LINE-intersect*, which determines if two given line segments intersect. As shown below, we originally defined two lines to be intersecting if they have a point in common, which we feel is a very intuitive and readable definition.

$$\text{LINE-intersect}(l_1, l_2) = \exists p: \text{POINT}, \text{LINE-contains}(l_1, p) \wedge \text{LINE-contains}(l_2, p)$$

Our effort to translate it into an executable definition resulted in an almost unreadable definition shown in Figure 2. In the executable definition, if the two lines are orthogonal, then they

```

LINE-intersect( $l_1$  ,  $l_2$ ) =
  if (LINE-horizontal( $l_1$ )  $\wedge$  LINE-vertical( $l_2$ ))
  then (LINE-contains(POINT-create(POINT-X-coordinate(LINE-to( $l_2$ )),
                                     POINT-Y-coordinate(LINE-to( $l_1$ ))),
                                      $l_1$ )
         $\wedge$  LINE-contains(POINT-create(POINT-X-coordinate(LINE-to( $l_2$ )),
                                     POINT-Y-coordinate(LINE-to( $l_1$ ))),
                                      $l_2$ ))
  elseif (LINE-horizontal( $l_2$ )  $\wedge$  LINE-vertical( $l_1$ ))
  then (LINE-contains(POINT-create(POINT-X-coordinate(LINE-to( $l_1$ )),
                                     POINT-Y-coordinate(LINE-to( $l_2$ ))),
                                      $l_1$ )
         $\wedge$  LINE-contains(POINT-create(POINT-X-coordinate(LINE-to( $l_1$ )),
                                     POINT-Y-coordinate(LINE-to( $l_2$ ))),
                                      $l_2$ ))
  else
    (LINE-contains(LINE-from( $l_1$ ),  $l_2$ )  $\vee$  LINE-contains(LINE-to( $l_1$ ),  $l_2$ )
      $\vee$  LINE-contains(LINE-from( $l_2$ ),  $l_1$ )  $\vee$  LINE-contains(LINE-to( $l_2$ ),  $l_1$ ))

```

Figure 2: Executable Definition of LINE-intersect

intersect if the point formed by taking the x-intercept of the vertical line and the y-intercept of the horizontal line lies on both lines. If the lines are not orthogonal, then they intersect (or overlap) if and only if one of the end-points of one line lies on the other line. We abandoned our quest for executability soon after as it violated our primary goal of readability. Our experiments agree with the hypothesis of Bidoit et al. [?] that readability can be achieved if it is a primary goal, and that the pursuit of readability may require relaxing the requirement of executability.

4.2.3.5 Abstractness A specification should describe only the properties of the system (*what* the system does) while omitting the implementation details (*how* it is done). This allows the implementors of the system to choose from a wide range of implementations that the specification can be refined into. A specification cluttered with implementation details might make the specification hard to understand and also eliminate the best possible implementation from being considered. Therefore, abstractness was one of our primary goals. We found that executability and abstractness are often incompatible with each other. The *LINE-intersect* example in Section 4.2.3.4 is a good example of this incompatibility. We also found that while abstractness usually enhances readability, it can also make a specification hard to understand in some cases. In the AGL specification, we had to compromise between readability and abstractness in the choice of range type for *AGL-nodes* and *AGL-edges*. The order of addition of edges and nodes to AGL is important; different orders of addition of nodes and edges result in different graph layouts (of the same graph). The initial version of the AGL specification reflected this conceptual notion: *AGL-nodes* and *AGL-edges* had sequences as their range types; they preserved the order in which nodes and edges were added to a graph. But later we realized that this was an unnecessary restriction. While it is necessary to know the order of the addition of nodes and edges in order to layout the graph, the order information is not a property of the edges and the nodes of the graph. Instead, it is a property of the graph itself. *AGL-nodes* and *AGL-edges* should therefore have sets as their range types. The information about the order in which nodes and edges are added to AGL is reflected

in the order in which the operations *AGL-add-node* and *AGL-add-edge* appear in an expression of type AGL and can be inferred from the properties of those operations. But the use of sets as the choice of range types for nodes and edges makes the specification harder to understand since the ordering information is now buried in the properties of *AGL-add-node* and *AGL-add-edge*. In this case, we opted for abstractness over readability since readability can be improved to some extent with comments.

4.2.3.6 Operation-Oriented Development versus Object-Oriented Development Specification methods tend to focus on either a system's operations or its data [?]. An operation-oriented method identifies the system's key functions and specifies their properties while describing data through simplistic models. An object-oriented (data-oriented) method identifies the key types of objects in the system, the operations performed on the objects, and their properties. An object-oriented approach is best suited to develop modular algebraic specifications because sorts and operations in an algebraic specification correspond naturally to classes and methods. The object-oriented approach is also well-suited to specification construction by reuse and for constructing reusable specification components. Reusable specification components can be used repeatedly after verifying them once, thus resulting in fast and reliable development of specifications. We can build modular specifications that are easy to read and maintain by using reusable specification components. We therefore used an object-oriented approach to develop our specification. We obtained reusable specification components for basic types such as natural numbers, real numbers, and Booleans, and developed all the other components shown in Appendix ?? . From our experience, we feel that an object-oriented approach can be used very effectively in certain domains such as AGL where objects and operations can be clearly identified.

4.2.3.7 Negotiations and Trade-offs A primary reason for constructing formal specifications is to evaluate whether an implementation of the required software system is constructible. If the specification is not satisfiable, no working implementations of the specification can be constructed. Thus constructing a formal specification can help detect impossible (combinations of) requirements early in the software requirements process. The AGL aesthetic criteria are an example of this. It is not possible to construct an implementation of an AGL system that will independently minimize edge lengths as well as edge crossings. For example, Figure ?? shows two possible edges between nodes n_1 and n_2 . Edge e_1 has the minimum length and edge e_2 has the minimum number of edge crossings. But there can be no edge with the length of e_1 and the number of crossings of e_2 . An implementation of AGL must therefore achieve some combination of the two aesthetic criteria instead of achieving them independently. In such cases, it becomes necessary to negotiate with the client and convince the client of the impossibility of meeting all requirements. The client can then choose from a set of realistically achievable requirements combinations. Negotiations and trade-offs such as these are an inevitable part of the specification building process [?, ?].

The specification developer also has to make other kinds of trade-offs while developing specifications. The definition of the *LINE-intersect* operation in Section 4.2.3.4 illustrates a trade-off between executability and readability. The choice of range types for *AGL-nodes* and *AGL-edges* as discussed in Section 4.2.3.5 demonstrates a trade-off between abstractness and readability. Such trade-offs are common during specification construction and should be managed by making the objectives of the specification construction process explicit.

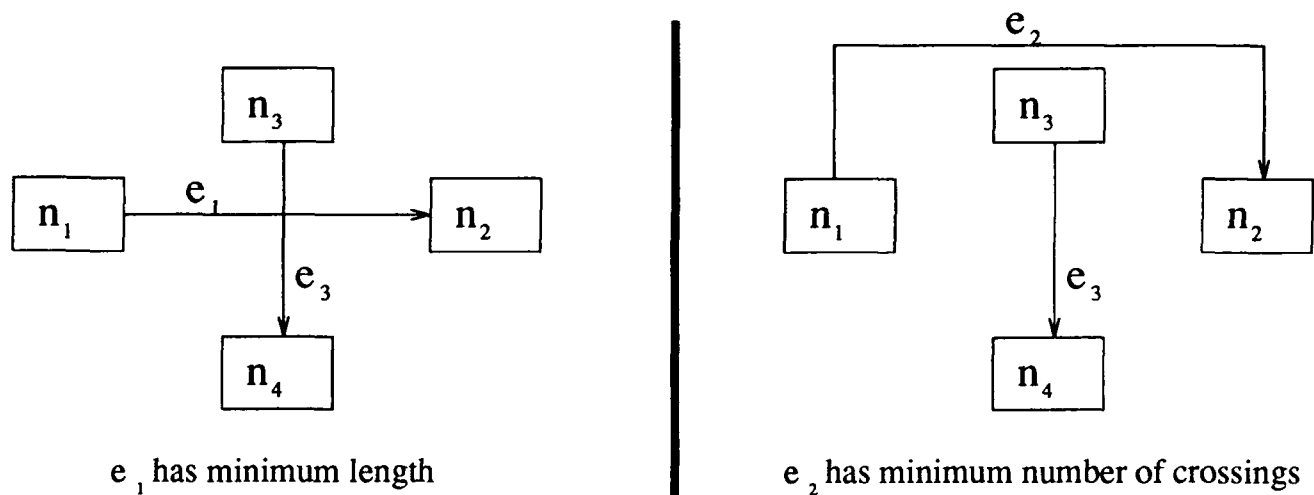


Figure 3: Two possible edges between nodes n_1 and n_2

4.2.4 Conclusion and Directions for Future Work

We found the exercise of writing a detailed formal specification for the problem of aesthetic graph layout interesting and useful in helping us gain a deeper understanding of the problem and the process of formalizing specifications. The main difficulty we encountered during the formalization process was the lack of necessary reusable components. For example, we could have greatly benefited from the existence of a repository with components for specifying point, line, rectangle, and graph. In general, we would also want mechanisms to retrieve components based on desired properties, and then adapt and compose such components to tailor them to the current problem. We found that modifying specification components often results in introducing new errors into the specifications due to the interdependencies between specification components. This problem could be substantially alleviated if the specification developer could identify all the potentially affected parts of the specification after each change to the specification. It may be possible to build a formal transformational system to assist in identifying all such affected parts. A specification developer would use the formal transformations provided by such a system to systematically introduce changes into the specification at the semantic level instead of using a text editor. Feather and Johnson [?, ?] introduced the notion of evolution transformations as a means of elaborating specifications. We are extending that notion to design formal evolution transformations that can preserve certain properties while changing others. These formal evolution transformations can also be used to identify changed portions of a specification. We are designing these formal evolution transformations as operations in a theory of specification reuse [?].

4.2.5 The AGL Specification

4.2.5.1 AGL Specification Component spec AGL =

sorts: AGL

based on: REAL, BOOLEAN, POINT, LINE, LINE-SEQUENCE, NODE, EDGE,
 NODE-SET, EDGE-SET, NAT, LINE-SET

operations:

```

AGL-create:  $\rightarrow$  AGL
AGL-add-node:  $\text{POINT} \times \text{REAL} \times \text{REAL} \times \text{AGL} \rightarrow \text{AGL}$ 
AGL-delete-node:  $\text{NODE} \times \text{AGL} \rightarrow \text{AGL}$ 
AGL-add-edge:  $\text{NODE} \times \text{NODE} \times \text{AGL} \rightarrow \text{AGL}$ 
AGL-delete-edge:  $\text{EDGE} \times \text{AGL} \rightarrow \text{AGL}$ 
AGL-nodes:  $\text{AGL} \rightarrow \text{NODE-SET}$ 
AGL-edges:  $\text{AGL} \rightarrow \text{EDGE-SET}$ 
AGL-add-non-overlapping-node:  $\text{POINT} \times \text{REAL} \times \text{REAL} \times \text{AGL} \rightarrow \text{AGL}$ 
AGL-add-edge-set:  $\text{EDGE-SET} \times \text{AGL} \rightarrow \text{AGL}$ 
AGL-delete-edge-set:  $\text{EDGE-SET} \times \text{AGL} \rightarrow \text{AGL}$ 
AGL-potential-edge-segments:  $\text{AGL} \times \text{NODE} \times \text{NODE} \times \text{LINE-SEQUENCE} \rightarrow \text{BOOLEAN}$ 
AGL-most-aesthetic-edge-segments:  $\text{AGL} \times \text{NODE} \times \text{NODE} \times \text{LINE-SEQUENCE} \rightarrow \text{BOOLEAN}$ 
AGL-no-other-node-intersections:  $\text{AGL} \times \text{NODE} \times \text{NODE} \times \text{LINE-SEQUENCE} \rightarrow \text{BOOLEAN}$ 
AGL-edge-segments-cost:  $\text{AGL} \times \text{LINE-SEQUENCE} \rightarrow \text{REAL}$ 
AGL-edge-length-cost-factor:  $\rightarrow \text{REAL}$ 
AGL-edge-bends-cost-factor:  $\rightarrow \text{REAL}$ 
AGL-edge-crossings-cost-factor:  $\rightarrow \text{REAL}$ 

```

axioms:

```

AGL-nodes(AGL-create) =  $\emptyset$ 
AGL-edges(AGL-create) =  $\emptyset$ 
AGL-nodes(AGL-add-node(p, x, y, g)) =
  if  $\exists n_1: \text{NODE}, n_1 \in \text{AGL-nodes}(g) \wedge \text{NODE-node-overlap}(n, n_1)$ 
  then AGL-nodes(g)
  else AGL-nodes(g)  $\cup \{n\}$ 
  where n:  $\text{NODE} = \text{NODE-create}(p, x, y)$ 
AGL-edges(AGL-add-node(p, x, y, g)) =
  if  $\exists n_1: \text{NODE}, n_1 \in \text{AGL-nodes}(g) \wedge \text{NODE-node-overlap}(n, n_1)$ 
  then AGL-edges(g)
  else AGL-edges(AGL-add-edge-set
    (s, AGL-add-non-overlapping-node(p, x, y, AGL-delete-edge-set(s, g))))
  where
    s:  $\text{EDGE-SET} = \{e: \text{EDGE} \mid e \in \text{AGL-edges}(g) \wedge \text{NODE-edge-intersect}(n, e)\}$ 
    n:  $\text{NODE} = \text{NODE-create}(p, x, y)$ 
AGL-nodes(AGL-add-non-overlapping-node(p, x, y, g)) =
  AGL-nodes(g)  $\cup \{\text{NODE-create}(p, x, y)\}$ 
AGL-edges(AGL-add-non-overlapping-node(p, x, y, g)) = AGL-edges(g)
AGL-add-edge-set( $\emptyset$ , g) = g
AGL-add-edge-set( $\{e\} \cup s$ , g) =
  AGL-add-edge-set(s, AGL-add-edge(EDGE-source(e), EDGE-target(e), g))
AGL-delete-edge-set( $\emptyset$ , g) = g
AGL-delete-edge-set( $\{e\} \cup s$ , g) = AGL-delete-edge-set(s, AGL-delete-edge(e, g))
AGL-nodes(AGL-delete-node(n, g)) = AGL-nodes(g) -  $\{n\}$ 
AGL-edges(AGL-delete-node(n, g)) =
   $\{e: \text{EDGE} \mid e \in \text{AGL-edges}(g) \wedge \text{EDGE-source}(e) \neq n \wedge \text{EDGE-target}(e) \neq n\}$ 
AGL-nodes(AGL-delete-edge(e, g)) = AGL-nodes(g)
AGL-edges(AGL-delete-edge(e, g)) = AGL-edges(g) -  $\{e\}$ 
AGL-nodes(AGL-add-edge( $n_1, n_2, g$ )) = AGL-nodes(g)
AGL-edges(AGL-add-edge( $n_1, n_2, g$ )) = AGL-edges(g)  $\cup \{\text{EDGE-create}(n_1, n_2, g)\}$ 
AGL-most-aesthetic-edge-segments(g,  $n_1, n_2, \text{EDGE-segments}(\text{EDGE-create}(n_1, n_2, g))$ ) = true
AGL-most-aesthetic-edge-segments(g,  $n_1, n_2, s$ ) =
  AGL-potential-edge-segments(g,  $n_1, n_2, s$ )
   $\wedge \forall s_1: \text{LINE-SEQUENCE}, \text{AGL-potential-edge-segments}(g, n_1, n_2, s_1)$ 
     $\Rightarrow (\text{AGL-edge-segments-cost}(g, s) \leq \text{AGL-edge-segments-cost}(g, s_1))$ 
AGL-potential-edge-segments(g,  $n_1, n_2, s$ ) =
  ( $\forall l: \text{LINE}, l \in s \Rightarrow (\text{LINE-horizontal}(l) \vee \text{LINE-vertical}(l))$ )
   $\wedge \text{LINE-SEQUENCE-connected}(s) \wedge \text{NODE-connects}(n_1, n_2, s)$ 
   $\wedge \text{AGL-no-other-node-intersections}(g, n_1, n_2, s)$ 
   $\wedge \sim \text{LINE-SEQUENCE-overlap}(s, \text{set-to-sequence}(\text{NODE-boundaries}(n_1)))$ 
   $\wedge \sim \text{LINE-SEQUENCE-overlap}(s, \text{set-to-sequence}(\text{NODE-boundaries}(n_2)))$ 
   $\wedge \forall e: \text{EDGE}, e \in \text{AGL-edges}(g) \Rightarrow \sim \text{LINE-SEQUENCE-overlap}(s, \text{EDGE-segments}(e))$ 
AGL-no-other-node-intersections(g,  $n_1, n_2, s$ ) =

```


$$\begin{aligned} & \forall n: \text{NODE}, (n \in \text{AGL-nodes}(g) \wedge n \neq n_1 \wedge n \neq n_2) \\ & \quad \Rightarrow \neg \text{LINE-SEQUENCE-intersect}(s, \text{set-to-sequence}(\text{NODE-boundaries}(n))) \\ \text{AGL-edge-segments-cost}(g, s) = & \\ & ((\sum_{l \in s} \text{LINE-length}(l)) * \text{AGL-edge-length-cost-factor}()) \\ & + ((|s| - 1) * \text{AGL-edge-bends-cost-factor}()) \\ & + ((\sum_{e \in \text{AGL-edges}(g)} \text{LINE-SEQUENCE-intersections}(s, \text{EDGE-segments}(e))) \\ & \quad * \text{AGL-edge-crossings-cost-factor}()) \end{aligned}$$

end spec AGL

4.2.5.2 NODE Specification Component spec NODE =

sorts: NODE

based on: REAL, BOOLEAN, POINT, LINE, LINE-SEQUENCE, LINE-SET, EDGE

operations:

NODE-create: POINT \times REAL \times REAL \rightarrow NODE
 NODE-position: NODE \rightarrow POINT
 NODE-X-extent: NODE \rightarrow REAL
 NODE-Y-extent: NODE \rightarrow REAL
 NODE-top-left-corner: NODE \rightarrow POINT
 NODE-top-right-corner: NODE \rightarrow POINT
 NODE-bottom-left-corner: NODE \rightarrow POINT
 NODE-bottom-right-corner: NODE \rightarrow POINT
 NODE-boundaries: NODE \rightarrow LINE-SET
 NODE-node-overlap: NODE \times NODE \rightarrow BOOLEAN
 NODE-edge-overlap: NODE \times EDGE \rightarrow BOOLEAN
 NODE-edge-intersect: NODE \times EDGE \rightarrow BOOLEAN
 NODE-connects: NODE \times NODE \times LINE-SEQUENCE \rightarrow BOOLEAN
 NODE-has-point-on-boundary: NODE \times POINT \rightarrow BOOLEAN

axioms:

NODE-position(NODE-create(p, x, y)) = p
 NODE-X-extent(NODE-create(p, x, y)) = x
 NODE-Y-extent(NODE-create(p, x, y)) = y
 NODE-top-left-corner(n) =
 POINT-create(POINT-X-coordinate(NODE-position(n)) - NODE-X-extent(n)/2,
 POINT-Y-coordinate(NODE-position(n)) + NODE-Y-extent(n)/2)
 NODE-top-right-corner(n) =
 POINT-create(POINT-X-coordinate(NODE-position(n)) + NODE-X-extent(n)/2,
 POINT-Y-coordinate(NODE-position(n)) + NODE-Y-extent(n)/2)
 NODE-bottom-left-corner(n) =
 POINT-create(POINT-X-coordinate(NODE-position(n)) - NODE-X-extent(n)/2,
 POINT-Y-coordinate(NODE-position(n)) - NODE-Y-extent(n)/2)
 NODE-bottom-right-corner(n) =
 POINT-create(POINT-X-coordinate(NODE-position(n)) + NODE-X-extent(n)/2,
 POINT-Y-coordinate(NODE-position(n)) - NODE-Y-extent(n)/2)
 NODE-boundaries(n) =
 {LINE-create(p₁, p₂), LINE-create(p₂, p₃),
 LINE-create(p₃, p₄), LINE-create(p₄, p₁)}
 where
 p₁ = NODE-top-left-corner(n)
 p₂ = NODE-top-right-corner(n)
 p₃ = NODE-bottom-right-corner(n)
 p₄ = NODE-bottom-left-corner(n)
 NODE-node-overlap(n₁, n₂) =
 LINE-SEQUENCE-intersect(set-to-sequence(NODE-boundaries(n₁)),

```

                                set-to-sequence(NODE-boundaries(n2)))
NODE-edge-intersect(n, e) =
    LINE-SEQUENCE-intersect(set-to-sequence(NODE-boundaries(n)), EDGE-segments(e))
NODE-edge-overlap(n, e) =
    LINE-SEQUENCE-overlap(set-to-sequence(NODE-boundaries(n)), EDGE-segments(e))
NODE-has-point-on-boundary(n, p) =
    ∃ l: LINE, l ∈ NODE-boundaries(n) ∧ LINE-contains(l, p)
NODE-connects(n1, n2, s) =
    NODE-has-point-on-boundary(n1, LINE-from(first(s)))
    ∧ NODE-has-point-on-boundary(n2, LINE-to(last(s)))

```

end spec NODE

4.2.5.3 EDGE Specification Component spec EDGE =

sorts: EDGE

based on: LINE, LINE-SEQUENCE, AGL

operations:

```

EDGE-create: NODE × NODE × AGL → EDGE
EDGE-source: EDGE → NODE
EDGE-target: EDGE → NODE
EDGE-segments: EDGE → LINE-SEQUENCE
/* Standard set and sequence operations for sets and sequences of edges */

```

axioms:

```

EDGE-source(EDGE-create(n1, n2, g)) = n1
EDGE-target(EDGE-create(n1, n2, g)) = n2
/* Standard set and sequence axioms for sets and sequences of edges */

```

end spec EDGE

4.2.5.4 LINE Specification Component spec LINE =

sorts: LINE

based on: REAL, BOOLEAN, POINT, NAT

operations:

```

LINE-create: POINT × POINT → LINE
LINE-from: LINE → POINT
LINE-to: LINE → POINT
LINE-horizontal: LINE → BOOLEAN
LINE-vertical: LINE → BOOLEAN
LINE-length: LINE → REAL
LINE-intersect: LINE × LINE → BOOLEAN
LINE-overlap: LINE × LINE → BOOLEAN

```

axioms:

```

LINE-from(LINE-create(p1, p2)) = p1
LINE-to(LINE-create(p1, p2)) = p2
LINE-horizontal(l) = (POINT-Y-coordinate(LINE-from(l)) = POINT-Y-coordinate(LINE-to(l)))
LINE-vertical(l) = (POINT-X-coordinate(LINE-from(l)) = POINT-X-coordinate(LINE-to(l)))
LINE-length(l) = POINT-distance(LINE-from(l), LINE-to(l))

```

```

LINE-intersect( $l_1, l_2$ ) =  $\exists p$ : POINT, LINE-contains( $l_1, p$ )  $\wedge$  LINE-contains( $l_2, p$ )
LINE-overlap( $l_1, l_2$ ) =
   $\exists p_1, p_2$ : POINT,  $p_1 \neq p_2 \wedge$  LINE-contains( $l_1, p_1$ )
     $\wedge$  LINE-contains( $l_1, p_2$ )  $\wedge$  LINE-contains( $l_2, p_1$ )  $\wedge$  LINE-contains( $l_2, p_2$ )
LINE-contains( $l, p$ ) =
  (( $(y - y_1) * (x_2 - x_1) = (x - x_1) * (y_2 - y_1)$ ))  $\wedge$  ( $x_1 \leq x \leq x_2$ )  $\wedge$  ( $y_1 \leq y \leq y_2$ )
  where
    x = POINT-X-coordinate(p)
    y = POINT-Y-coordinate(p)
    x1 = POINT-X-coordinate(LINE-from( $l$ ))
    y1 = POINT-Y-coordinate(LINE-from( $l$ ))
    x2 = POINT-X-coordinate(LINE-to( $l$ ))
    y2 = POINT-Y-coordinate(LINE-to( $l$ ))

```

end spec LINE

4.2.5.5 LINE-SEQUENCE Specification Component spec LINE-SEQUENCE =

sorts: LINE-SEQUENCE

based on: REAL, BOOLEAN, POINT, NAT, LINE

operations:

```

/* Standard sequence operations for sequences of lines */
LINE-SEQUENCE-connected: LINE-SEQUENCE  $\rightarrow$  BOOLEAN
LINE-SEQUENCE-overlap: LINE-SEQUENCE  $\times$  LINE-SEQUENCE  $\rightarrow$  BOOLEAN
LINE-SEQUENCE-intersect: LINE-SEQUENCE  $\times$  LINE-SEQUENCE  $\rightarrow$  BOOLEAN
LINE-SEQUENCE-intersections: LINE-SEQUENCE  $\times$  LINE-SEQUENCE  $\rightarrow$  NAT

```

axioms:

```

/* Standard sequence operations for sequences of lines */
LINE-SEQUENCE-overlap( $s_1, s_2$ ) =
   $\exists l_1, l_2$ : LINE,  $l_1 \in s_1 \wedge l_2 \in s_2 \wedge$  LINE-overlap( $l_1, l_2$ )
LINE-SEQUENCE-intersect( $s_1, s_2$ ) =
   $\exists l_1, l_2$ : LINE,  $l_1 \in s_1 \wedge l_2 \in s_2 \wedge$  LINE-intersect( $s_1, s_2$ )
LINE-SEQUENCE-intersections( $s_1, s_2$ ) =
   $\sum_{l_1 \in s_1} \sum_{l_2 \in s_2}$  if LINE-intersect( $l_1, l_2$ ) then 1 else 0
LINE-SEQUENCE-connected([]) = TRUE
LINE-SEQUENCE-connected([ $l$ ]) = true
LINE-SEQUENCE-connected(prepend( $l_1$ , prepend( $l_2$ ,  $s$ ))) =
  LINE-to( $l_1$ ) = LINE-from( $l_2$ )  $\wedge$  LINE-SEQUENCE-connected(prepend( $l_2$ ,  $s$ ))

```

end spec LINE-SEQUENCE

4.2.5.6 POINT Specification Component spec POINT =

sorts: POINT

based on: REAL

operations:

```

POINT-create: REAL  $\times$  REAL  $\rightarrow$  POINT
POINT-X-coordinate: POINT  $\rightarrow$  REAL

```

```
POINT-Y-coordinate: POINT  $\rightarrow$  REAL
POINT-distance: POINT  $\times$  POINT  $\rightarrow$  REAL
```

axioms:

```
POINT-X-coordinate(POINT-create(a,b)) = a
POINT-Y-coordinate(POINT-create(a,b)) = b
POINT-distance( $p_1, p_2$ ) =
  sqrt(sqr(POINT-X-coordinate( $p_1$ ) - POINT-X-coordinate( $p_2$ ))
    + sqr(POINT-Y-coordinate( $p_1$ ) - POINT-Y-coordinate( $p_2$ )))
```

end spec POINT

4.3 A Theory of Composition Operations for Software Reuse

The Concise Oxford Dictionary defines *composition* as “the act of putting together”, “formation”, or “construction”. In software reuse, composition operations are used to put together two or more software components to build new software components. Examples of composition include combining the functionalities of different components, identifying the functionality of one component that is not also provided by another component, or selecting functionality that is common to two components.

Observe that composition of reusable components to obtain the desired software artifact is not an easy task. The major problem is in verifying that the result of composition is indeed the desired artifact. Even if the properties of reusable components are known in advance, it is usually difficult to determine the properties of the software artifact produced by composition. This is because composition operations are usually not defined to produce definite identifiable changes at the semantic level. For software reuse to be more practical, we need to be able to easily determine the properties of the result of composition. This requirement has the following implications:

- We should have a repository of reusable components with known properties, and
- We should have a collection of composition operations with known behaviors.

In this paper, we describe certain composition operations that can be used to compose *algebraic specification* components to produce *software specifications* with certain guaranteed properties. In addition to illustrating the use of these operations to compose reusable components, we also show how they can be used for *reactive* reusable component design. All the composition operations described here have been fully implemented.

This paper is organized as follows. In the next section, we present a motivating example. The example depicts the activities in a typical specification reuse scenario. In Section ??, we carefully design four composition operations and ensure that they are defined uniquely. We present some properties of these operations in Section ??. In Section ??, we discuss related work, and finally present some directions for future research.

4.3.1 A Motivating Example

This example is based on the Improved Many on Many (IMOM) electronic combat decision aid, version 4.0 [?]. IMOM gives electronic combat planners a quick, accurate way of portraying the combat scenario to aid in charting aircraft flight paths, determining aircraft visibility to radar,

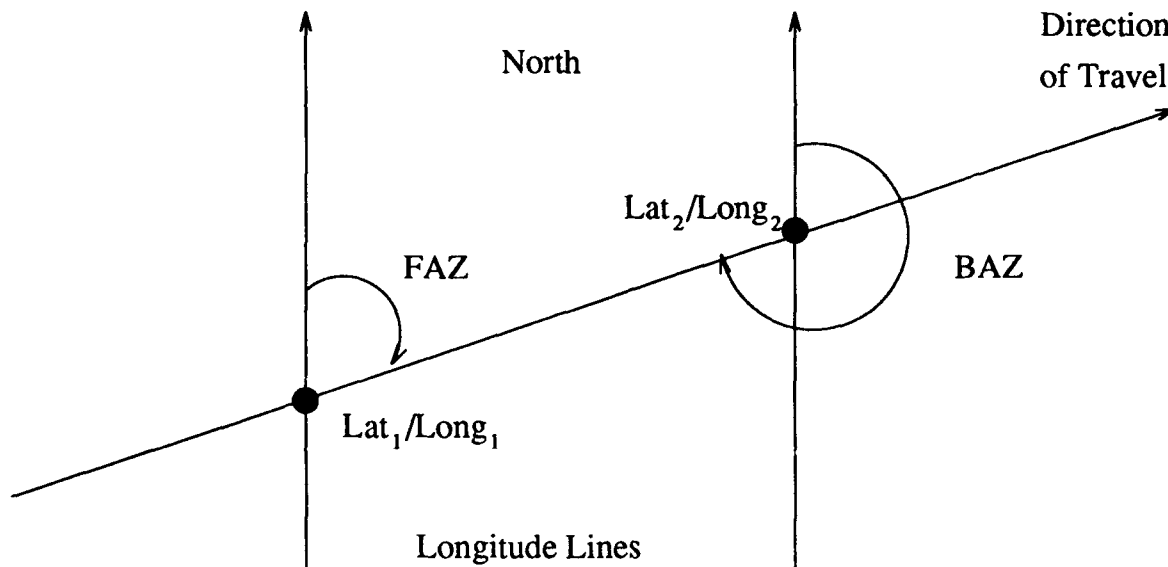


Figure 4: FAZ is the forward azimuth angle and BAZ is the backward azimuth angle. Note that FAZ and BAZ are not supplementary angles as the longitudes are not parallel lines.

radar jamming, and so on. We choose a small part of the IMOM system functionality for our example.

One part of the IMOM functionality is the computation of *forward* and *backward azimuths*. Forward and backward azimuths are the angles between the flight path of an aircraft and the intersecting longitude lines at two given points on the aircraft's flight path. Forward azimuth is the angle made by the longitude at the first given point with the direction of travel (between the two latitude/longitude points) measured in a clockwise manner. Backward azimuth is the same but measured at the second latitude/longitude point (see Figure ??). Azimuths (i.e., global directions of travel) can be used to determine relationships between the aircraft's flight path and the intersecting longitudes. These relationships are useful in finding geometric solutions to certain other problems in the IMOM domain.

Another part of the IMOM functionality is the computation of the *elevation* of a place, which is necessary for determining line of sight and terrain masking. The IMOM system uses a database that stores the elevations of various points on the earth's surface indexed by the latitude and longitude. Given the latitude and longitude of a place, the IMOM system either returns its elevation, or the elevation of a nearest point in the database.

To compute azimuths and elevation at given points, the IMOM system needs to compute the latitude and longitude of each of those points. The IMOM system computes the latitude/longitude of a new terrain point using the known latitude/longitude of a nearby point, the angular distance of the new point from the known point, and the angle made by the arc connecting the two points with true north (see Figure ??).

The motivating example is a scenario based on a small part of the IMOM specification. We focus on the specification of azimuths and elevation. We assume that we have two specification components initially:

1. **Azimuth-1.** This component specifies the computation of azimuths. The computation of

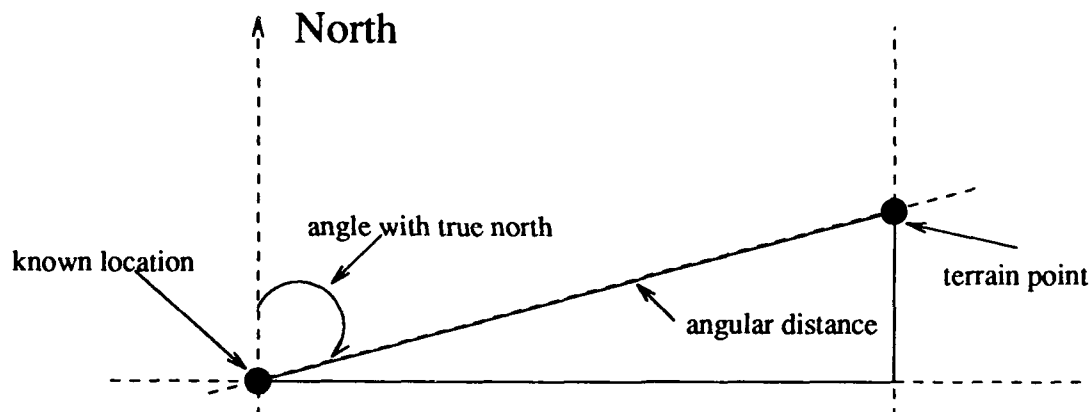


Figure 5: The latitude and longitude of the terrain point are computed using the location of the known point, angular distance from the known point, and angle made by arc connecting the points with true north.

azimuths depends on the earth's curvature at each of the two given locations. Since the earth is not a perfect sphere, this component specifies the computation of earth's curvature at each of the two points, and uses the exact curvature to specify azimuths. The two points are specified relative to a known nearby location. This component also specifies the computation of latitude and longitude of a given point.

2. **Elevation-1.** This component specifies the computation of the elevation at a given point. If the point is in the database, its exact elevation is obtained. Otherwise, the average of the elevations of its four nearest neighbors is used. The location of the point is again specified relative to a known nearby location. This component also specifies the computation of latitude and longitude of a given point.

To construct the IMOM specification, we want a single component that calculates both the azimuths and the elevation. We compose the two components, **Azimuth-1** and **Elevation-1**, to get a new component, **Azim-Elev-2**, that specifies computations of azimuths and elevations, and has a single copy of the latitude/longitude computation (assuming that their computations in the two components are the same). We call this operation *join*.

The latitude/longitude computation is needed in almost all parts of the IMOM system. We create a reusable component specifying the latitude/longitude computation by extracting the common parts of **Azimuth-1** and **Elevation-1** to get a component, **lat-long-3**. We call this operation *meet*. Thus, we use the meet operation to create reusable components *reactively*.

In **Azim-Elev-2**, the determination of the earth's curvature at each point is computationally expensive. Moreover, such accuracy is not needed in azimuth calculation. We, therefore, modify the specification to assume that the earth's curvature is constant. This modification is performed either manually or using evolution transformations [?, ?, ?]. The result is a new component **Azim-Elev-4**.

For computing elevation at a point, it is sufficient to take its nearest neighbor's elevation instead of the average of the elevations of its four nearest neighbors. Again, we make this change in **Azim-Elev-2** either manually or using evolution transformations to get **Azim-Elev-5**. Note that the changes to **Azim-Elev-2** to get **Azim-Elev-4** and **Azim-Elev-5** are performed in

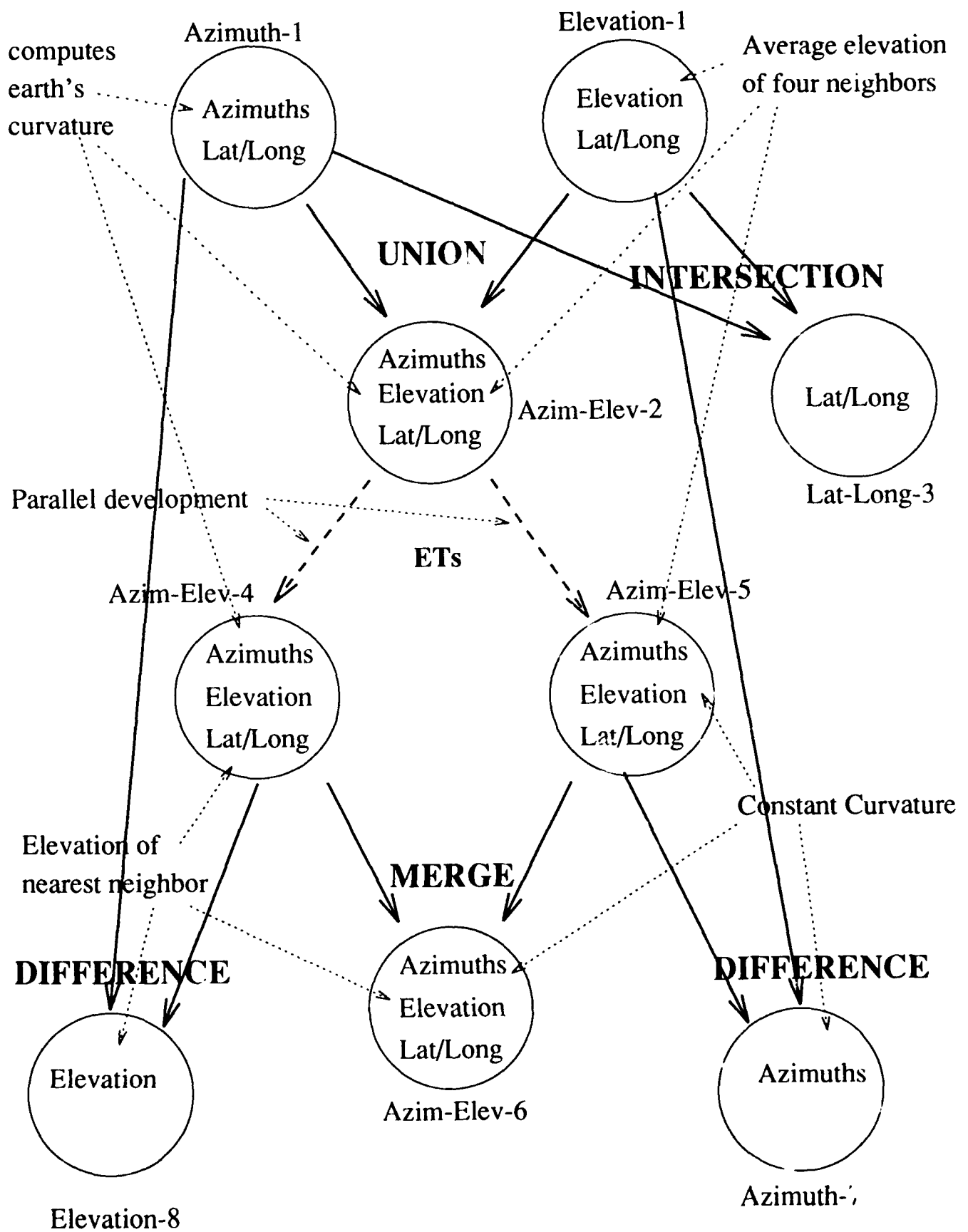


Figure 6: The graph of specification evolution and design of new reusable components.

parallel.

Next, we want a component that uses the constant curvature assumption for azimuth computation (as in **Azim-Elev-4**) and the nearest neighbor for elevation computation (as in **Azim-Elev-5**). So we perform a *merge* operation on **Azim-Elev-2**, **Azim-Elev-4**, and **Azim-Elev-5**; the part of the specification in **Azim-Elev-2** that is unchanged in the two modified components is combined with the changed parts from the modified components (provided the meanings of the changed parts do not conflict) to get **Azim-Elev-6**.

We finally decide that the new azimuth calculation in **Azim-Elev-4** and the new elevation calculation in **Azim-Elev-5** are important. We want to extract and store them as separate reusable components. So we get a new azimuth component, **Azimuth-7**, by subtracting **Azim-Elev-4** from **Elevation-1**. We call this operation *difference*. Similarly, we get a new elevation component, **Elevation-8**, by taking the difference of **Azim-Elev-5** and **Azimuth-1**.

The complete specification evolution graph is shown in Figure ??.

4.3.2 The Composition Operations

In this section we will briefly look at the design criteria, the design process, and the actual definitions of the following composition operations:

- *Meet*. The meet of two specification components is a component that has information common to both the components.
- *Difference*. The difference of two specification components (given in a specific order) is a component that contains information in the first specification component that is not in the second specification component.
- *Join*. Join takes two components and produces a third component that has all the information from both the components.
- *Merge*. Merge acts on three components: a base component, and two components that have been evolved from the base component. The result of merge is a component that has information common to all three components, along with the new or changed information in the two evolved components. Note that join is a special case of merge, equivalent to merge with an empty base component.

As argued in the introduction, we want operations whose behaviors are known: that is, operations whose application results in a definite change, and which guarantee certain properties of the results they produce given inputs with certain properties.

We, therefore, have the following design criteria:

- The composition operations should be analyzable for determining the semantic changes they produce; hence they should be formal.
- They should produce definite, predictable changes to guide their selection and application, and assist in validation of components produced.
- With input components that have certain properties (e.g., consistency), the operations should guarantee certain properties of the result (including preservation of important properties such as consistency).

```

spec STACK
  signature
    sorts
      stack, nat, boolean
    operations
      zero:  $\rightarrow$  nat
      succ: nat  $\rightarrow$  nat
      error:  $\rightarrow$  nat
      true:  $\rightarrow$  boolean
      false:  $\rightarrow$  boolean
      create:  $\rightarrow$  stack
      push: stack, nat  $\rightarrow$  stack
      pop: stack  $\rightarrow$  stack
      top: stack  $\rightarrow$  nat;
      isempty: stack  $\rightarrow$  boolean;
    constructors
      zero, succ, true, false, create, push
  axioms
    pop(create) = create
    pop(push(s, n)) = s
    top(create) = error
    top(push(s, n)) = n
    isempty(create) = true
    isempty(push(s, n)) = false
end

```

Figure 7: Stack specification.

In order to be able to describe the composition operations, we first present our simplified specification language and its algebraic semantics. Then we define the four composition operations and ensure that the definitions guarantee uniqueness.

4.3.2.1 Specification Language In the simplified specification language we have chosen for this discussion, a specification is a theory consisting of a set of sorts, a set of operations, a set of constructors, and a set of equational axioms [?]. The set of constructors is a subset of the set of operations. We assume that the specifications are such that every ground term (i.e., a term without any variables) can be rewritten using the axioms of the specification to a term containing only constructor symbols. The set of sorts, the set of constructors, and the set of operations are together called the *signature* of the specification.

Definition 3 *Specification.* A specification SP consists of a pair $\langle \Sigma, E \rangle$ where

- $\Sigma = \langle S, F, C \rangle$ is the *signature* of SP . S is the set of *sorts*, F is the set of *operations*, and $C \subseteq F$ is the set of *constructors*;
- E is the set of *equational axioms* of SP . ■

An example of a specification appears in Figure ??.

The specification language has an algebraic semantics. An algebra is a collection of sets and a collection of functions on these sets. An algebraic model of a specification is defined as follows:

Definition 4 Algebraic Model. Given a specification $SP = \langle \Sigma, E \rangle$ where $\Sigma = \langle S, F, C \rangle$, a model of SP consists of an algebra $A = \langle A_S, F_A \rangle$ where

- $A_S = \{A_s \mid s \in S\}$ is a collection of sets called the *carrier sets* of the algebra;
- $F_A = \{f_A \mid f \in F\}$ is a collection of *functions* of the algebra, such that if the rank of f is $s_1, \dots, s_n \rightarrow s$, then f_A is a function from $A_{s_1} \times \dots \times A_{s_n}$ to A_s ;
- A satisfies all the axioms in E . ■

For a more detailed description of algebraic semantics, including the meaning of satisfiability of axioms, see Wirsing's paper [?].

Given a specification in this specification language there is a class of algebraic models for the specification. If the specification satisfies certain conditions [?], there exists a distinguished algebra (up to isomorphism) in the class of algebraic models. This distinguished algebra, called the *initial algebra*, has a unique homomorphism to each of the other algebraic models of the specification. We restrict the discussion here to specifications that are guaranteed to have an initial algebra. Henceforth in this proposal when we say the "meaning of a specification" we mean its initial algebra.

Next we define the notion of a subreduct. We will use this notion in stating the properties of most adapt operations.

Definition 5 Subreduct. A subreduct B of an algebra A is an algebra such that all carrier sets in B are also carrier sets in A and all functions in B are also functions in A . ■

We are now ready to define the four adapt operations mentioned earlier.

Definition 6 Meet. Given specifications SP_1 and SP_2 with meanings M_1 and M_2 respectively, the meet of SP_1 and SP_2 is a specification SP with meaning M such that

- M is isomorphic to a subreduct of M_1 ;
- M is isomorphic to a subreduct of M_2 ; and
- If M' is any algebra that is isomorphic to a subreduct of M_1 and a subreduct of M_2 , then either
 - a subreduct of M is isomorphic to M' , or
 - M is not a subreduct of M' . ■

Informally, the meet of two specifications SP_1 and SP_2 can be described as the specification corresponding to the largest isomorphic subreducts of the meanings M_1 and M_2 of the two specifications. Meet, as defined above, is not unique because the set of subreducts of M_1 that are isomorphic to some subreducts of M_2 ordered by the subreduct relation need not have a *maximum* element. However, using the axiom of choice, we can show that this set of subreducts \mathcal{S} does have *maximal* elements. Each maximal element of \mathcal{S} can be considered to be the meaning of the meet of the two given specifications.

As one of our design criteria is to define each composition operation uniquely, we can make the meaning of meet unique by including only certain interesting subreducts in \mathcal{S} , and by restricting

the isomorphisms that are considered. These restrictions make \mathcal{S} a complete lattice. As a complete lattice has a top element, \mathcal{S} has a unique largest subreduct that is the meaning of the meet of the two specifications.

The first restriction is to consider only a certain class of subreducts, called *constructor subreducts* for inclusion in \mathcal{S} .

Definition 7 Constructor Subreduct. Let A be the initial algebra (i.e., meaning) of a specification SP . A subreduct B of A is called a constructor subreduct of A if for each carrier set B_s in B corresponding to some sort s in SP , all the functions f_A in A corresponding to constructors f of sort s in SP are also in B . ■

We restrict the isomorphisms by specifying a partial mapping σ between the signatures of the two specifications. From a requirements standpoint, the mapping σ specifies the parts of the two signatures that are intended to be the same. Thus, this mapping rules out frivolous matches, and also ensures that if two subreducts are isomorphic, there is a unique isomorphism between them. The following definition formally defines the mapping σ .

Definition 8 Signature Morphism. A signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$ from signature $\Sigma_1 = \langle S_1, F_1, C_1 \rangle$ to signature $\Sigma_2 = \langle S_2, F_2, C_2 \rangle$ consists of a tuple $\langle \sigma_S, \sigma_F \rangle$ where

- $\sigma_S : S_1 \rightarrow S_2$ is a function;
- $\sigma_F : F_1 \rightarrow F_2$ is a function such that if the rank of $f_1 \in F_1$ is $s_1, \dots, s_n \rightarrow s$ and $\sigma_F(f_1) = f_2$ then the rank of f_2 is $\sigma_S(s_1), \dots, \sigma_S(s_n) \rightarrow \sigma_S(s)$.

σ is a partial signature morphism if either σ_S , or σ_F , or both are partial functions. ■

If for two signatures Σ_1 and Σ_2 , the signature morphism $in : \Sigma_1 \rightarrow \Sigma_2$ is the canonical embedding that takes each sort or function symbol in Σ_1 to the same sort or function symbol in Σ_2 , then in is called the *inclusion* signature morphism.

If $\sigma : \Sigma_1 \rightarrow \Sigma_2$ is a partial signature morphism, then the set of symbols of Σ_1 for which σ is defined is called the *domain* of σ . The set of symbols of Σ_2 which are the images of some symbols in Σ_1 is called the *codomain* of σ .

Using this definition of the partial signature morphism σ , we can formally define the restricted isomorphisms between subreducts as follows:

Definition 9 σ -Isomorphism. Let $\sigma : \Sigma_1 \rightarrow \Sigma_2$ be a one-to-one partial signature morphism, where $\Sigma_1 = \langle S_1, F_1, C_1 \rangle$ and $\Sigma_2 = \langle S_2, F_2, C_2 \rangle$ are the signatures of specifications SP_1 and SP_2 , respectively. Let A and B be the initial models of SP_1 and SP_2 , respectively. Then an isomorphism $h : A' \rightarrow B'$, where A' and B' are subreducts of A and B respectively, is a σ -isomorphism if the following conditions hold:

- if h maps the carrier set A'_{s_1} in A' corresponding to the sort $s_1 \in S_1$ to the carrier set B'_{s_2} in B' corresponding to the sort $s_2 \in S_2$, then σ must map s_1 to s_2 ;
- if h maps the function $f_{1,A'}$ in A' corresponding to the operation $f_1 \in F_1$ to the function $f_{2,B'}$ in B' corresponding to the operation $f_2 \in F_2$, then σ must map f_1 to f_2 .

If $h : A' \rightarrow B'$ is a σ -isomorphism, then A' and B' are said to be σ -isomorphic. ■

If SP_1 and SP_2 are two specifications whose meanings are σ -isomorphic, then we say that SP_1 and SP_2 themselves are σ -isomorphic and we denote this by $SP_1 \cong_\sigma SP_2$.

Applying the above definition of σ -isomorphisms to constructor subreducts of the meanings of two specifications, we can prove the following theorem:

Theorem 1 *Given two specifications $SP_1 = \langle \Sigma_1, E_1 \rangle$ and $SP_2 = \langle \Sigma_2, E_2 \rangle$ and a one-to-one partial signature morphism σ from Σ_1 to Σ_2 , the collection of constructor subreducts of SP_1 that are σ -isomorphic to some constructor subreducts of SP_2 form a complete lattice under the subreduct ordering.*

Proof sketch. If $h : A' \rightarrow B'$ is a σ -isomorphism between constructor subreducts A' and B' , then h is unique. This can be proved by considering terms, and using induction on the depth of terms.

Let \mathcal{P} be the set of all constructor subreducts of SP_1 such that each of them is isomorphic to some constructor subreduct of SP_2 . In order to prove that \mathcal{P} is a complete lattice it is enough to show that it is non-empty, that it has a bottom element, and that every subset of it has a least upper bound [?].

\mathcal{P} is clearly non-empty as the empty constructor subreduct is in it and is isomorphic to the empty constructor subreduct of SP_2 . The empty constructor subreduct is also the bottom element of \mathcal{P} ².

For any subset $\mathcal{Q} \subseteq \mathcal{P}$, the union of all constructor algebras contained in \mathcal{Q} is the least upperbound of \mathcal{Q} . This can be proved by showing that the σ -isomorphisms for all the constructor subreducts in \mathcal{Q} can be put together to get a σ -isomorphism for the union of all constructor subreducts in \mathcal{Q} , which is thus a member of \mathcal{P} .

End of proof sketch. ■

Henceforth we denote this lattice \mathcal{P} of subreducts of SP_1 with respect to $\sigma : \Sigma_1 \rightarrow \Sigma_2$ as $\mathcal{L}(SP_1, SP_2, \sigma)$.

Now we can modify our earlier definition of meet so that meet is *unique* up to isomorphism.

Definition 10 *Meet.* Let $SP_1 = \langle \Sigma_1, E_1 \rangle$ and $SP_2 = \langle \Sigma_2, E_2 \rangle$ be specifications with meanings M_1 and M_2 respectively and σ be a signature morphism from Σ_1 to Σ_2 . The meet of SP_1 and SP_2 (denoted $SP_1 \wedge_\sigma SP_2$) with respect to σ is a specification $SP = \langle \Sigma, E \rangle$ where the meaning of SP , M , is isomorphic to the top element of $\mathcal{L}(SP_1, SP_2, \sigma)$ and there exists an inclusion signature morphism $in : \Sigma \hookrightarrow \Sigma_1$. ■

At this point, we make two further simplifying assumptions to limit the scope of this discussion. These assumptions allow us to talk about the results of the operations of the theory without worrying about the signature morphisms involved. Our first assumption is that the partial signature morphism σ is restricted to the symbols that are common to the two signatures, that is,

$$\text{domain}(\sigma) = \text{range}(\sigma) = \Sigma_1 \cap \Sigma_2 \quad (1)$$

Such a signature morphism is called a *name signature morphism*. With this assumption, we will denote the meet of two specifications SP_1 and SP_2 by $SP_1 \wedge SP_2$.

Our second assumption is that the two given specifications are *total σ -isomorphic* as defined below:

²Note that we did not define algebras to have a non-empty collection of carrier sets

Definition 11 Total σ -Isomorphism. Let $\sigma : \Sigma_1 \rightarrow \Sigma_2$ be a one-to-one partial signature morphism, where $\Sigma_1 = \langle S_1, F_1, C_1 \rangle$ and $\Sigma_2 = \langle S_2, F_2, C_2 \rangle$ are the signatures of specifications SP_1 and SP_2 , respectively. Let A and B be the initial models of SP_1 and SP_2 , respectively. Then SP_1 and SP_2 are total σ -isomorphic if there exists an isomorphism $h : A' \rightarrow B'$, where A' and B' are subreducts of A and B respectively, such that

- h is a σ -isomorphism;
- if σ maps sort s_1 in S_1 to sort s_2 in S_2 , then the carrier set corresponding to s_1 is in A' and the carrier set corresponding to s_2 is in B' ;
- if σ maps the operation f_1 in F_1 to the operation f_2 in F_2 , then the function corresponding to f_1 is in A' and the function corresponding to f_2 is in B' . ■

In the rest of the discussion, we will only consider specifications that are total σ -isomorphic with respect to their name signature morphisms. For total σ -isomorphic specifications $SP_1 = \langle \Sigma_1, E_1 \rangle$ and $SP_2 = \langle \Sigma_2, E_2 \rangle$, the signature of their meet $SP_1 \wedge SP_2$ is $\Sigma_1 \cap \Sigma_2$.

Using the definition of meet and the assumptions stated above, we can now uniquely define two other composition operations, **difference** and **join**. The difference between two specifications, SP_1 and SP_2 , is the same as the difference between the first specification and the meet of the two specifications. Intuitively we can obtain the difference SP_3 thus: From the meaning of the first specification, remove the part that is isomorphic to the meaning of the meet of the two given specifications, and then pick the smallest constructor subreduct of the meaning of the first specification that contains the remaining meaning. In general, the meet of SP_2 and SP_3 could be non-empty because some sorts or operations of SP_1 that are not in SP_2 may depend on some other sorts or operations that are common to both SP_1 and SP_2 . To ensure that SP_3 has a minimal meet with SP_2 , its meaning must be the smallest algebra with all the above properties. To show that this notion of difference is uniquely defined, we will need the fact that the constructor subreducts of this difference form a complete lattice.

Theorem 2 *Let M be the meaning of a specification SP , L be the lattice of constructor subreducts of M and A be a constructor subreduct of M . The set of constructor subreducts of M whose union with A is M itself forms a complete lattice under the subreduct ordering.*

Proof: Similar to the proof of Theorem ??.

We will use $\mathcal{L}(SP, A)$ to denote the lattice stated in Theorem ??.

We can now define **difference** uniquely as follows:

Definition 12 Difference. Let $SP_1 = \langle \Sigma_1, E_1 \rangle$ and $SP_2 = \langle \Sigma_2, E_2 \rangle$ be specifications with meanings M_1 and M_2 , respectively. Let SP_1 and SP_2 be total σ -isomorphic with respect to the name signature morphism σ between their signatures. Let $SP_3 = \langle \Sigma_3, E_3 \rangle$, with meaning M_3 , be the meet of SP_1 and SP_2 . Then the difference between SP_1 and SP_2 (denoted $SP_1 \ominus SP_2$) is a specification $SP = \langle \Sigma, E \rangle$ with meaning M such that M is isomorphic to the bottom element of $\mathcal{L}(SP_1, M_3)$ and there exists an inclusion signature morphism $in : \Sigma \hookrightarrow \Sigma_1$. ■

The join of two specifications SP_1 and SP_2 is a specification with all of SP_1 and SP_2 , but with only one copy of the meet of SP_1 and SP_2 .

Definition 13 Join. Let $SP_1 = \langle \Sigma_1, E_1 \rangle$ and $SP_2 = \langle \Sigma_2, E_2 \rangle$ be total σ -isomorphic specifications with meanings M_1 and M_2 respectively, and let σ be the name signature morphism from Σ_1 to Σ_2 . Let SP_3 , with meaning M_3 , be the meet of SP_1 and SP_2 . Then the join of SP_1 and SP_2 (denoted $SP_1 \vee SP_2$) is a specification SP with meaning M such that $M = M' \vee M''$ with \vee denoting the union of algebras and where

- M' is isomorphic to M_1 ;
- M'' is isomorphic to M_2 ; and
- $M' \wedge M''$ is isomorphic to M_3 .

Furthermore, there exist inclusion signature morphisms $in_1 : \Sigma_1 \hookrightarrow \Sigma$ and $in_2 : \Sigma_2 \hookrightarrow \Sigma$, and the signature of SP is $\Sigma = \Sigma_1 \vee \Sigma_2$. ■

Without the total σ -isomorphism assumption, join is a partial operation because two sorts or operations in SP_1 and SP_2 that are intended to be isomorphic (via σ) may not be isomorphic. Note that the result of the join operation is uniquely defined because of the unique result of the meet operation.

We can now define the *merge* operation. Merge is a partial operation that takes three specifications, SP_1 , SP_2 , and SP_3 , and merges them to produce a new specification SP_4 . SP_1 is the base specification; SP_2 and SP_3 are specifications that are obtained by modifying SP_1 . SP_4 , if defined, contains the following sorts and operations:

- all the sorts and operations in SP_1 left unchanged in SP_2 and SP_3 ;
- all the sorts and operations in SP_2 that are either new or changed as compared to SP_1 ;
- all the sorts and operations in SP_3 that are either new or changed as compared to SP_1 .

Definition 14 Merge. The merge of three specifications, SP_1 , SP_2 , and SP_3 , if defined, is $SP = (SP_2 \wedge SP_3) \vee ((SP_3 \ominus SP_1) \vee (SP_2 \ominus SP_1))$

If we did not have the assumption about total σ -isomorphisms, merge would be a partial operation. Without total σ -isomorphisms, the various joins in the definition of merge may not be defined. The uniqueness of merge is guaranteed by the uniqueness of \wedge , \vee , and \ominus .

Some properties of the three composition operations defined above are presented in the next section.

4.3.3 Properties of the Composition Operations

To provide guidance in the use of the composition operations defined in the previous section, we need to know how they interact. In this section, we study their various properties. We again assume all specifications to be total σ -isomorphic where σ is the name-isomorphism.

1. (Commutativity) $SP_1 \wedge SP_2 \cong SP_2 \wedge SP_1$

Since SP_1 and SP_2 are total σ -isomorphic, SP_2 and SP_1 are also total σ -isomorphic. Therefore, $\mathcal{L}(SP_1, SP_2, \sigma)$ is isomorphic to $\mathcal{L}(SP_2, SP_1, \sigma^{-1})$ because each element of the former lattice, being a constructor subreduct of SP_1 and SP_2 , is also in the latter lattice, and vice versa.

2. (Commutativity) $SP_1 \vee SP_2 \cong SP_2 \vee SP_1$

This follows directly from the definition of union of algebras and commutativity of \wedge .

3. (Idempotency) $SP \wedge SP \cong SP$

The top element of $\mathcal{L}(SP, SP, \sigma)$ is the meaning of SP itself because σ is a total signature morphism, and the meaning of SP is a constructor subreduct of itself.

4. (Idempotency) $SP \vee SP \cong SP$

This follows from the fact that the union operation on algebras is idempotent.

5. (Associativity) $(SP_1 \wedge SP_2) \wedge SP_3 \cong SP_1 \wedge (SP_2 \wedge SP_3)$

Let SP_l denote the l.h.s. and SP_r denote the r.h.s. of this equation. Let x be an element (i.e., carrier set or function) of the meaning of SP_l . Therefore, it is isomorphic to some element y of the meaning of $SP_1 \wedge SP_2$ as well as to some element z in the meaning of SP_3 . Therefore, x is isomorphic to some element in the meaning of all three specifications. Therefore it belongs to the meaning of SP_r . Therefore, the meaning of SP_l is a subreduct of the meaning of SP_r . Similarly, it can be shown that the meaning of SP_r is a subreduct of the meaning of SP_l . Therefore, SP_l and SP_r are isomorphic.

6. (associativity) $(SP_1 \vee SP_2) \vee SP_3 \cong SP_1 \vee (SP_2 \vee SP_3)$

Let SP_l denote the l.h.s. and SP_r denote the r.h.s. of this equation. Let x be an element (i.e., carrier set or function) of the meaning of SP_l . Therefore it belongs to the meaning of either SP_3 or $SP_1 \vee SP_2$. Suppose it belongs to SP_3 . Therefore, it also belongs to the meaning of $SP_2 \vee SP_3$ and hence to the meaning of SP_r . Similarly, we can show that elements belonging to the meanings of SP_1 or SP_2 also belong to the meaning of SP_r . Therefore, meaning of SP_l is a subreduct of the meaning of SP_r . Similarly, it can be shown that the meaning of SP_r is a subreduct of the meaning of SP_l . Therefore, SP_l and SP_r are isomorphic.

7. (Distributivity) $SP_1 \wedge (SP_2 \vee SP_3) \cong (SP_1 \wedge SP_2) \vee (SP_1 \wedge SP_3)$

Let SP_l denote the l.h.s. and SP_r denote the r.h.s. of this equation. Let x be an element (i.e., carrier set or function) of the meaning of SP_l . Therefore it belongs to the meaning of SP_1 as well as $SP_2 \vee SP_3$. Therefore, it also belongs to the meaning of either SP_2 or SP_3 . Suppose it belongs to the meaning of SP_2 . Therefore, it belongs to the meaning of $SP_1 \wedge SP_2$, and hence to the meaning of SP_r . Similarly, we can show that x belongs to the meaning of SP_r if it belongs to the meaning of SP_3 . Therefore, SP_l is a subreduct of the meaning of SP_r . On the other hand, if x belongs to the meaning of SP_r , then it belongs to the meaning of either $SP_1 \wedge SP_2$ or $SP_1 \wedge SP_3$. If it belongs to the meaning of $SP_1 \wedge SP_2$, then it also belongs to the meanings of both SP_1 as well as SP_2 , and hence to the meaning of SP_l . Similarly, if it belongs to $SP_1 \wedge SP_3$, then it belongs to the meaning of SP_l . Therefore, the meaning of SP_r is a subreduct of the meaning of SP_l . Therefore, SP_l and SP_r are isomorphic.

8. (Distributivity) $SP_1 \vee (SP_2 \wedge SP_3) \cong (SP_1 \vee SP_2) \wedge (SP_1 \vee SP_3)$

Let SP_l denote the l.h.s. and SP_r denote the r.h.s. of this equation. Let x be an element (i.e., carrier set or function) of the meaning of SP_l . Therefore it belongs to the meaning of SP_1 or $SP_2 \wedge SP_3$. If it belongs to the meaning of SP_1 , it also belongs to the meanings of $SP_1 \vee SP_2$ and $SP_1 \vee SP_3$. On the other hand, if x belongs to the meaning of both SP_2

and SP_3 , then again it belongs to the meaning of SP_r . Therefore, meaning of SP_l is a subreduct of the meaning of SP_r . On the other hand, if x belongs to SP_r , then it belongs to to the meanings of both $SP_1 \vee SP_2$ and $SP_1 \vee SP_3$. If it belongs to the meaning of SP_1 , then it also belongs to the meaning of SP_l . On the other hand, if it does not belong to the meaning of SP_1 , then it must belong to the meanings of both SP_2 and SP_3 , and hence to the meaning of SP_l . Therefore, the meaning of SP_r is a subreduct of the meaning of SP_l . Therefore, SP_l and SP_r are isomorphic.

9. $SP \ominus SP \cong \perp$

Since $SP \wedge SP$ is SP itself, the smallest subreduct of SP such that its join with $SP \wedge SP$ is SP itself is the bottom element.

4.3.4 Related Work

Horowitz, Prins, and Reps [?] discuss one operation for composing programs. This operation helps integrate three programs: a base program and two variants of the base program. The result of integration contains the information in the base program together with the information in the variants that has changed from the base. During integration of changed information from the variants it is ensured that they do not conflict. We adapt this operation to get *merge* that works on software specifications, and add three more compose operations, *meet*, *join*, and *difference*. Meet combines the information in two components into one after ensuring that the information does not conflict (thus meet is a special case of merge wherein the base component is empty). Join extracts the information common to two specification components, and difference selects information present in the first component but not in the second.

In the survey on algebraic semantics for specifications, Wirsing [?] discusses work on specification building operations. While the operations discussed there are designed at the semantic level, the only operation that resembles any of our composition operations is the *union* operation. This operation assumes that the symbols used in the two input components to be identical and produces a component containing information (i.e., axioms) present in either of the input components. Our goal in designing the join operation was for it to be more useful in practice; when performing the join of two components, it would help the specification developer to know if there is any conflict between the two.

Feather and Johnson [?, ?, ?] discuss evolution transformations (ETs): transformations that produce stereotypical changes in the specifications they are applied to. Most of the ETs are adaptation operations, helping in modifying the information contained in a component. Feather [?, ?] discusses how two components can be merged by using the sequences of evolution transformations that produced these components. While we agree that this is a good approach to use ultimately, we currently do not have adaptation operations that have been designed at the semantic level to produce specific changes in the properties of components. Furthermore, in practice, there are always manual changes performed together with the application of ETs to produce a desired software component. So there is a need for composition operations that do not rely solely on ET histories.

4.3.5 Conclusion

In this paper, we have presented four operations - meet, join, difference, and merge - for composing formal software specifications. These operations work at the semantic level and guarantee

certain properties of the result produced. The operations help in building new validated, reusable specification components, and also help in detecting any conflicts that may exist among the components being composed.

We know from our experience in implementing these operations that the ideas here can be extended to compose reusable components of software code written in functional languages. We need to investigate changes needed to make the operations useful for composing other forms of reusable components such as software designs or reusable components of code written in imperative or object-oriented languages. We also need to analyze the operations we have for completeness and design new operations if necessary.

We also need to design formal operations for adapting properties of individual specifications components. Then the interactions between various adapt and compose operations can be analyzed to determine properties that can be used to guide in specification reuse and development.

4.4 Implementation of Composition Operations

4.4.1 Design of Implementation Operations

In this section, we will briefly look at the design criteria, the design process, and the actual definitions of the following composition operations:

- *Meet*. The meet of two specification components is a component that has information common to both the components.
- *Difference*. The difference of two specification components (given in a specific order) is a component that contains information in the first specification component that is not in the second specification component.
- *Merge*. The merge operation is applied to three components: a base component, and two components that have been produced by modifying the base component. The result of merge is a component that has information common to all three components, along with the new or changed information in the two evolved components.
- *Join*. The join of two components is a component that has all the information from both the components. This is a special case of merge, equivalent to merge with an empty base component.

4.4.1.1 Design Criteria As argued in the introduction, we want operations whose behaviors are known: that is, operations whose application results in a definite change, and that guarantee certain properties of the results they produce given inputs with certain properties.

We, therefore, have the following design criteria:

- The composition operations should be analyzable for determining the semantic changes they produce; hence they should be formal.
- They should produce definite, predictable changes to assist in the validation of their results.
- With input components that have certain properties (e.g., consistency), the operations should guarantee certain properties of the result (including preservation of important properties such as consistency).

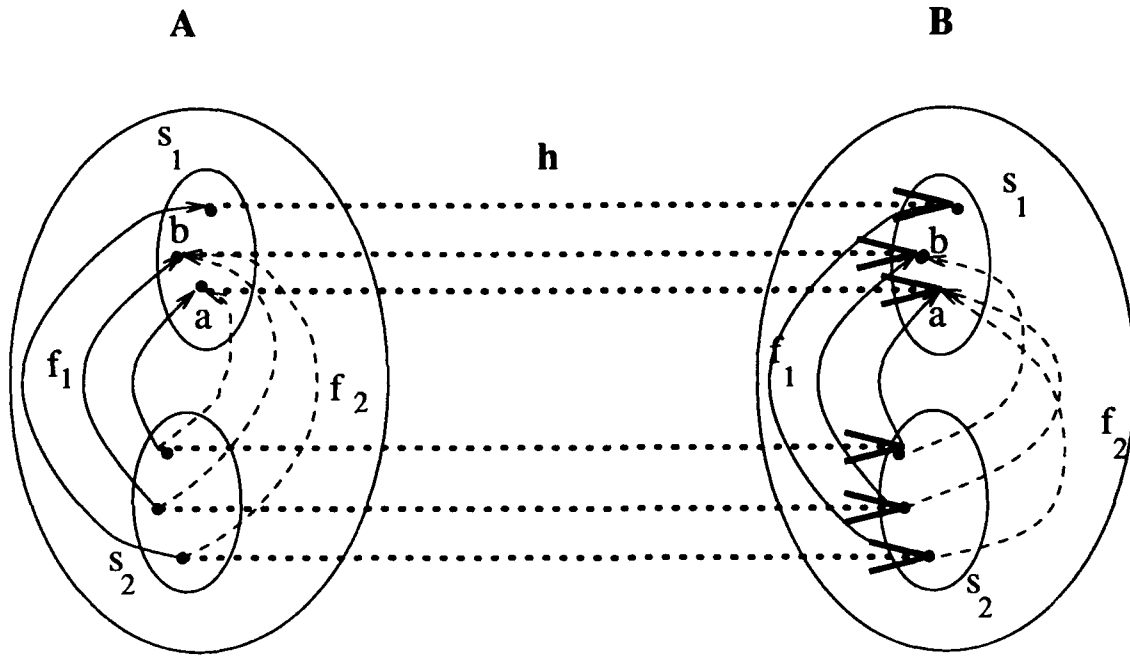


Figure 8: Two maximal meets of A and B exist: one whose meaning consists of the sets s_1 and s_2 with the operation f_1 , and another whose meaning has the two sets with the operation f_2 .

4.4.1.2 Design In this section, we discuss the design of the operations meet, difference, merge, and join, and analyze their behavior using algebraic semantics³. We assume all specifications to be equational. For conceptual clarity, we have attempted to present the ideas informally by reducing mathematical notation. The interested reader can find detailed proofs in [?].

4.4.1.2.1 Meet The meet of two specification components, SP_1 and SP_2 , denoted by $SP_1 \wedge SP_2$, is a component SP_3 such that the meaning of SP_3 is an algebra that is a subreduct⁴ of the meaning of SP_1 as well as that of SP_2 . Moreover, we want the meaning of SP_3 to be the largest such algebra⁵. The meaning of SP_3 must be a subreduct of the meanings of SP_1 and SP_2 so that SP_1 does not contain any information that is not in both SP_1 and SP_2 . Its meaning must be the largest such algebra so that SP_3 contains everything common to SP_1 and SP_2 .

Unfortunately, meet, defined as above, is not unique. Given specification components SP_1 and SP_2 , there can be several different specification components SP_3 that fit the definition. If we take the common subreducts of the meanings of SP_1 and SP_2 with the subreduct relation as the partial ordering among these algebras, we do not necessarily get a lattice. Thus, there may be several mutually incomparable maximal subreducts in this collection. For example, in Figure ??, if A and B are the meanings of the specification components SP_1 and SP_2 , respectively, then two maximal intersections exist. The meaning of one possible maximal intersection of SP_1 and SP_2 is

³That is, each specification has an algebra as its semantics. An algebra is a collection of sets together with functions on these sets. For instance, the set of natural numbers, the set of stacks of natural numbers, and functions create, push, pop, and top might be the algebraic semantics of a stack specification.

⁴A subreduct M of an algebra N is an algebra consisting of a subset of the sets in N and a subset of the functions in N . Of course, for each function in M , its domain and range sets should also be included in M .

⁵We use the subreduct relation for ordering algebras.

an algebra consisting of sets s_1 and s_2 with the function f_1 . The map h shown in the figure maps the elements of the sets to make f_1 compatible in the two algebras. But the same map makes f_2 incompatible.

If we change the map h to another map h' (not shown in the figure) that maps element a in A to b in B and b in A to a in B , then we get another maximal intersection consisting of sets s_1 and s_2 , and function f_2 . Clearly, this map makes the function f_1 from the two algebras incompatible.

Thus, the definition of meet violates our design criterion that the composition operations should produce definite changes. We, therefore, must further constrain the definition of meet.

In Figure ??, if we allow only the map h between the algebras (and exclude h' or any other map), then we will have only one maximal algebra consisting of the two sets and the function f_1 . This is precisely the restriction we impose.

Each of our specifications has a set of specially designated operations called *constructors*. We assume that the specifications are such that every ground term (i.e., a term without any variables) can be rewritten using the axioms of the specification to a term containing only constructor symbols. In other words, the specification is sufficiently complete [?, ?]. We now redefine the meet of two specifications to be a specification whose meaning is the largest *constructor subreduct*⁶ of the meanings of the two given specifications. We also provide a partial map, σ , between the signatures (sorts and operations) of SP_1 and SP_2 to specify parts of the two specification that are intended to be the same (from a requirements perspective). This map rules out frivolous matches, and it can be proved that the common constructor subreducts of SP_1 and SP_2 now form a lattice with a unique maximum subreduct (up to isomorphism). The map, h , between the meanings of the specifications is derived from σ . In Figure ??, if we assume that f_1 is a constructor, then we get a unique meet: an algebra consisting of the sets S_1 and S_2 , and the function f_1 . In the rest of this section, we assume that the partial maps, σ , providing a correspondence between various specifications under consideration are given by the user⁷.

The operation meet as defined above has the properties we desire. If the specifications SP_1 and SP_2 are internally consistent and sufficiently complete, then SP_3 has these properties, too. SP_3 is consistent because we take constructor subreducts, and SP_3 does not have anything that was not already in SP_1 and SP_2 . SP_3 is sufficiently complete because we take constructor subreducts.

4.4.1.2.2 Join Given two specifications SP_1 and SP_2 , the join of SP_1 and SP_2 , denoted by $SP_1 \vee SP_2$, is a specification component SP_3 such that the meanings of SP_1 and SP_2 are constructor subreducts of the meaning of SP_3 . Moreover, the meaning of SP_3 is the smallest such algebra. The meanings of SP_1 and SP_2 must be subreducts of the meaning of SP_3 so that SP_3 contains everything in SP_1 and SP_2 . It must be the the smallest such algebra so that it does not contain any information that is not in either SP_1 or SP_2 .

Join is a partial operation. Given two specification components and a map between their signatures, we may not be able to produce a join if there is a conflict. A conflict is said to occur when two sorts or operations that are intended to be semantically equivalent (according to the

⁶An algebra α is a constructor subreduct of the algebra β if α is a subreduct of β , and if for every carrier set in α , all the functions in β corresponding to the constructors of the sort corresponding to that carrier set are also in α .

⁷Actually, in our implementation, we construct a default map between specification components to map sorts and operations in one specification component to sorts and operations with the same name in the other specification component. The user can always override the default.

specified map) are not equivalent. If there is no such conflict, then the join operation should combine one copy of the common parts of the two specifications with the unique parts in the two specifications. Thus, we can construct the algebra corresponding to the result of join by taking the carrier sets and functions that are common to the meanings of the given specifications along with the sets and functions that are unique to the meanings of each of the two given specifications.

Join, as defined above, results in a unique specification because, as in meet, we only take constructor subreducts. Join preserves consistency since the result of join is defined only when there is no conflict between the two specifications. Join also preserves sufficient completeness as in the result of join (if it is defined), all the constructors of each sort exist in either both the given specifications, or in exactly one given specification.

4.4.1.2.3 Difference Given two specification components SP_1 and SP_2 , the difference between them, denoted by $SP_1 \ominus SP_2$, is a specification component SP_3 whose meaning is an algebra that is the smallest constructor subreduct of the meaning of SP_1 such that its join with $SP_1 \wedge SP_2$ is SP_1 itself. The meaning of SP_3 must be a subreduct of SP_1 so that SP_3 does not contain any information that is not in SP_1 . The join of SP_3 and $SP_1 \wedge SP_2$ must be SP_1 so that SP_3 contains all the information in SP_1 that is not also in SP_2 . In general, the meet of SP_2 and SP_3 could be non-empty because some sorts or operations of SP_1 that are not in SP_2 may depend on some other sorts or operations that are common to both SP_1 and SP_2 . To ensure that SP_3 has a minimal intersection with SP_2 , its meaning must be the smallest algebra with all the above properties.

The uniqueness of the definition of difference is guaranteed by the uniqueness of meet and join. We first find the unique meet of SP_1 and SP_2 , and then find the smallest constructor subreduct of the meaning of SP_1 such that its join with $SP_1 \wedge SP_2$ is SP_1 . It can be proved that a smallest such algebra always exists.

Difference preserves sufficient completeness as the meaning of the result of difference is a constructor subreduct of the meaning of SP_1 . Difference also preserves consistency as we only remove some parts from the equational specification SP_1 , which is known to be consistent.

4.4.1.2.4 Merge Merge is a partial operation that takes three specifications, SP_1 , SP_2 , and SP_3 , and merges them to produce a new specification SP_4 . SP_1 is the base specification; SP_2 and SP_3 are specifications that are obtained by modifying SP_1 . SP_4 , if defined, contains the following sorts and operations:

- all the sorts and operations in SP_1 left unchanged in SP_2 and SP_3 ;
- all the sorts and operations in SP_2 that are either new or changed as compared to SP_1 ;
- all the sorts and operations in SP_3 that are either new or changed as compared to SP_1 .

The result of merge is not defined if the three specifications *interfere* with one another. SP_1 , SP_2 , and SP_3 can interfere with one another in the following ways:

1. If some sort or operation in SP_1 is changed differently in both SP_2 and SP_3 .
2. If there are some semantically inequivalent sorts or operations in SP_2 and SP_3 that do not correspond to anything in SP_1 (by the partial maps provided between SP_1 and SP_2 , and SP_1 and SP_3), but that should correspond according to another map provided between SP_2 and SP_3 .

```

function ComputeDellat1(azimuth1 : angle1, dist1 : angle1) : angle1 =
    acos(cos(dist1) / cos(ComputeDellon1(dist1, azimuth1)))
function ComputeDellat2(azimuth2 : angle2, dist2 : angle2) : angle2 =
    acos(cos(dist2) / cos(ComputeDellon2(dist2, azimuth2)))
function ComputeDellon2(dist : angle2, azimuth : angle2) : angle2 =
    asin(sin(dist) * sin(azimuth))
function ComputeDellon1(dist : angle1, azimuth : angle1) : angle1 =
    asin(sin(dist) * sin(azimuth))

```

Figure 9: *ComputeDellat₁* and *ComputeDellat₂* are equivalent if *angle₁* and *angle₂* are equivalent

3. If there are new sorts or operations in SP_3 whose meaning depends on the meaning of a sort or operation (p) that has remained unchanged from SP_1 , but the meaning of the sort or operation corresponding to p has changed in SP_2 (p'). Because p' should be carried over to the result of merge (as it has changed from SP_1) and p should not, the new sorts or operations in SP_3 depending on p cannot be carried over. A symmetric case occurs if the new sorts or operations are in SP_2 .

The result of the merge operation is unique, and it preserves consistency and sufficient completeness.

4.4.2 Implementation

In this section, we describe our implementation of the composition operations. We have implemented the operations in a functional specification language called Refine [?]. The operations act on executable specification components that are also written in a subset of Refine. In this implementation, sorts are classes, constructors are functions, and non-constructors are either functions or maps between classes.

To implement the composition operations, we first made a few simplifying assumptions. All the operations described in the previous section depend on a notion of semantic equivalence between algebras. But proving that two specifications have the same meaning is theoretically unsolvable. So we use the syntactic equivalence in our implementation. We consider two sorts in any two given specifications to be equivalent if their corresponding constructors (implemented as functions) are equivalent. Any two functions, f_1 and f_2 , are equivalent if they have the same arguments and results, their bodies are syntactically equivalent, and any other sorts and functions that f_1 and f_2 depend on are also equivalent. For example, the functions *ComputeDellat₁* and *ComputeDellat₂* in Figure ?? are equivalent if the sorts *angle₁* and *angle₂* are equivalent.

Our current notion of equivalence is conservative, but it is sound; if our algorithms determine two specifications to be equivalent using our notion of equivalence, then the specifications can be proved to be semantically equivalent. On the other hand, there will be cases where specifications that are semantically equivalent will not be identified as equivalent by our algorithms.

Since equivalence of sorts depends on the equivalence of their constructors, and the equivalence of operations depends on the equivalence of the sorts and other operations that they depend on, constructing and matching dependency graphs is at the core of each of our composition operations. In the next few sections, we will first describe the algorithms for constructing and matching dependency graphs, and then describe how they are used in implementing each composition operation.

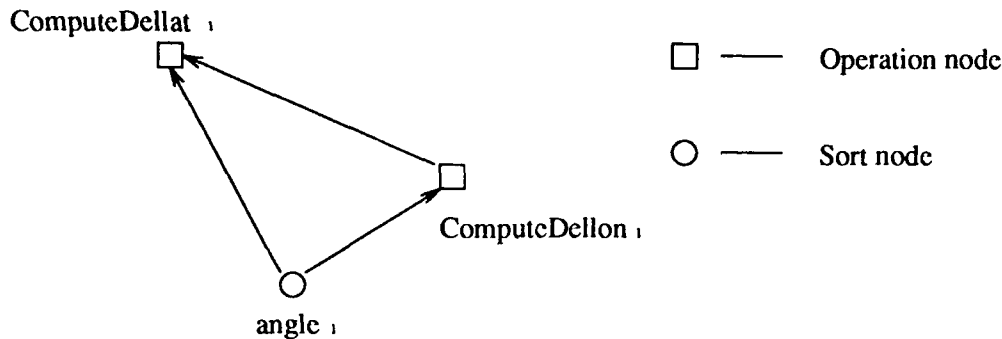


Figure 10: The function *ComputeDellat*₁ depends on the sort *angle*₁ and the function *ComputeDellon*₁.

4.4.2.1 Construction of Dependency Graphs A sort *s* in a specification is represented by a carrier set in the meaning of the specification. *s* depends on each of its constructors because changing the definition of any of its constructors may result in changing the elements of the carrier set corresponding to *s* in the meaning of the specification. An operation, on the other hand, is represented by a function in the algebra, and hence is dependent upon each sort in its rank (i.e., its input sorts and its output sort). In addition, changing the definition of an operation *foo* in the specification may result in changing the semantics of all other operations *bar* that depend on *foo*. Each operation, therefore, also depends on other operations used in its definition. The dependency graph for the function *ComputeDellat*₁ – defined in Figure ?? – is shown in Figure ??.

The algorithm for constructing the dependency graph is as follows⁸. For each sort and each operation in the specification, we add a node to the dependency graph. We add edges from the nodes corresponding to the constructors of each sort to the node corresponding to the sort that they construct. We then add edges from the nodes corresponding to the input and output sorts of each operation to the operation node. Finally, if an operation *foo* uses another operation *bar* in its definition, we add an edge from the node for *bar* to the node for *foo*.

4.4.2.2 Matching Dependency Graphs To determine the parts of the specifications that are equivalent, we match the dependency graphs. Matching helps us distinguish parts of two specifications that are equivalent from the parts of the specifications that may not be equivalent. We use a node coloring scheme to make this distinction. We also use node colors to distinguish nodes that are already processed from nodes yet to be processed. To match two dependency graphs, all nodes of both the graphs are first colored blue (blue indicates nodes that have not yet been processed). From the user-supplied map between the sorts and operations of the two specifications, we obtain a set of pairs of nodes that should be compared against each other⁹. We then examine each pair of blue nodes in this set to see if the corresponding sorts or operations are identical. For pairs of sort nodes, we check if their constructors can be paired up one-to-one for matching. If they do, then we color the two sort nodes amber, and add the pairs of constructor

⁸Actually, our specifications allow hidden sorts and hidden operations using an import/export mechanism. This adds further complexity to the algorithms, which we have omitted for conceptual clarity.

⁹Without this set, matching reduces to a subgraph isomorphism problem, which is NP-complete. Moreover, the match found using subgraph isomorphism may not be the match the user intended.

nodes to the set of node pairs to be processed, if they are not already there. A pair of amber nodes represent conditionally equivalent sorts or operations: they are equivalent if all the other sorts and operations that these nodes depend on are also equivalent. If the two nodes do not match, then we color them red, and follow their dependency edges in the graphs to color all the nodes that depend on these nodes red, as well. The color red represents possible mismatch. All the dependent nodes of any red nodes (i.e., nodes that can be reached from the red nodes by following the edges of the graph) are also colored red, since a mismatch among a node pair's dependents can result in a mismatch between the depending nodes, too.

For node pairs corresponding to the operations of the specifications, we check if they have similar ranks, and if their bodies are identical (except for use of other operation symbols in the body). If they are similar, we color both the operation nodes amber, and add any new pairs of sort or operation nodes from either the ranks or the body of the original operations to the set of node pairs to be processed. If we find a mismatch, we color those nodes red along with all the other nodes that can be reached from these operation nodes.

Finally, when there are no more unexamined (i.e., blue) node pairs in the set of node pairs to be processed, the matching process terminates. At termination, there may still be some amber nodes in the two graphs in addition to the unmatched blue, matched green, and mismatched red nodes. The presence of amber nodes at this stage indicates cyclic dependencies among these nodes. Since the amber nodes are conditionally equivalent, and they do not depend on any mismatched nodes, we finally color them green. At this point, the green nodes of the two graphs represent sorts or operations of the specifications that are semantically equivalent, the red nodes indicate sorts or operations that may not be semantically equivalent, and the blue nodes of each graph represent sorts and operations that do not have a counterpart in the other specification.

We can illustrate the graph matching process with the following example. Suppose that the function *ComputeDellat₂* in Figure ?? is defined as follows:

```
function ComputeDellat2(azimuth2 : angle2, dist2 : angle2) : angle2 =
  asin(sin(dist2) / cos(ComputeDellon2(dist2, azimuth2)))
```

If we try to match this new *ComputeDellat₂* function with the original *ComputeDellat₁* function from Figure ?? and match *ComputeDellon₁* with *ComputeDellon₂*, the functions *ComputeDellon₁* and *ComputeDellon₂* would match (assuming that the sorts *angle₁* and *angle₂* whose definitions have not been shown in the figure match), but the functions *ComputeDellat₁* and *ComputeDellat₂* would not match. The resulting colored graphs are shown in Figure ?. If the specification containing *ComputeDellat₂* had another function *foo* that did not correspond to any function in the specification containing *ComputeDellat₂* then the node for *foo* would be colored blue.

4.4.2.3 The Composition Operations The composition operations use dependency graph construction and match algorithms to detect similarities and differences between specifications, and to detect potential conflicts in the case of join and merge. They then create new dependency graphs corresponding to the specification of the result of these operations. The resulting specification can be reconstructed easily from this dependency graph. We will not discuss the reconstruction process in this paper.

4.4.2.3.1 Meet Given two specifications, we construct their dependency graphs and match them. We then identify the largest subgraph containing only the green colored nodes and the

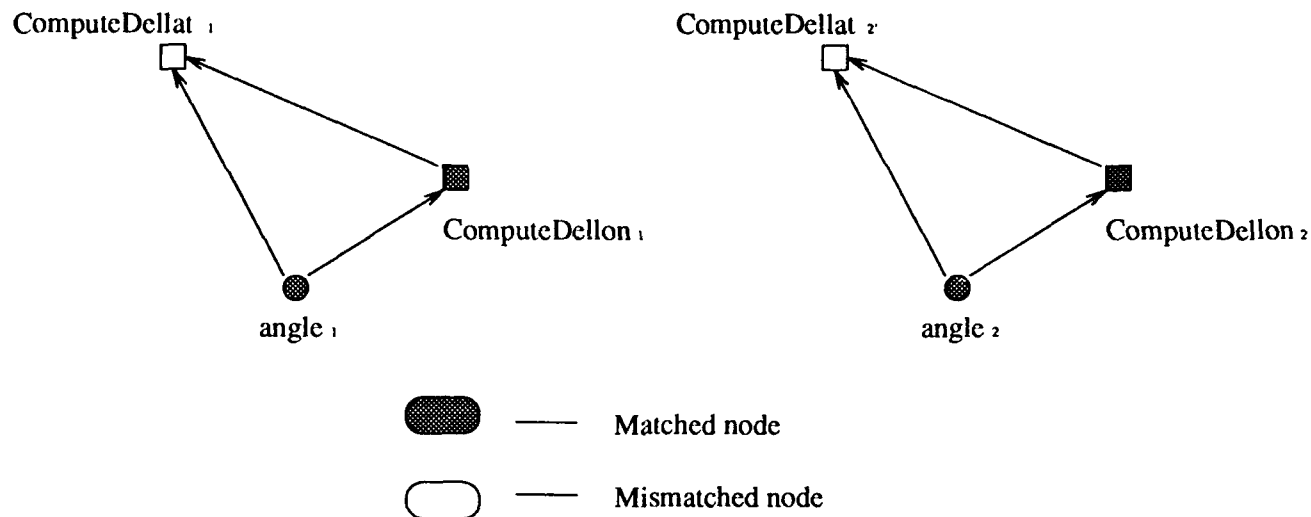


Figure 11: The dependency graphs of $ComputeDellat_1$ and $ComputeDellat_2$ are matched. $angle_1$ is known to match with $angle_2$. Therefore, $ComputeDellon_1$ matches with $ComputeDellon_2$.

edges between them from one of the dependency graphs (either one will work, as the subgraphs corresponding to the green nodes and the edges between them in the two graphs are isomorphic). This subgraph represents the semantic meet of the two input specifications.

4.4.2.3.2 Difference To determine the difference between two given specifications, we create and match the dependency graphs for the specifications. After matching, the green nodes in the two dependency graphs indicate the common elements between the two specifications. Therefore, to get the difference between the two specifications, we identify the largest subgraph of the first graph containing nodes that are either red (denoting sorts and operations in one specification that do not match with corresponding sorts and operations in the other specification) or blue (denoting sorts and operations in one specification that do not correspond to anything in the other specification), along with the edges between them. We then take the dependency closure of the nodes in this new subgraph. The nodes in the dependency closure represent sorts and operations on which the sorts and operations in the red and black nodes depend. Dependency closure is necessary, because the resulting specification would be incomplete without it.

4.4.2.3.3 Join The join of two specifications is a specification with information contained in either of the two given specifications. To get the the join of the specifications, we first ensure that there is no conflict between the two specifications. A conflict may exist if any corresponding sorts or operations in the two specifications cannot be proven to be equivalent. To determine existence of conflict, we first create and match the dependency graphs of the two specifications. The matching process colors corresponding nodes of the graphs red if their corresponding sorts or operations cannot be proved to be semantically equivalent. Therefore, presence of red nodes in the matched graphs indicates possibility of conflict.

If there are no red nodes in the two graphs after matching, the new dependency graph corresponding to the join is created by merging the two dependency graph. The graph merging process merges corresponding nodes if they are both green. Edges between nodes that are merged are

also merged. The resulting graph contains exactly one copy of the green nodes in the two original graphs and the edges between them, along with the unmatched nodes and edges from both the original graphs.

4.4.2.3.4 Merge The merge operation composes the base specification SP_1 with its two derivations SP_2 and SP_3 (obtained by modifying SP_1) to produce a new specification SP_4 . SP_4 contains the parts of SP_1 that have not changed in both SP_2 and SP_3 , and the new or changed parts of SP_2 together with the new or changed parts of SP_3 . Before we merge the specifications, however, we must ensure that there is no conflict among the specifications.

To detect conflicts, we first create two copies of the dependency graph of each specification. We then match the dependency graphs of SP_1 against the dependency graphs of SP_2 and SP_3 , and the dependency graph of SP_2 against the dependency graph of SP_3 . We then examine the matched graphs to identify the three possible cases of conflict enumerated in Section ?? as follows:

1. If a node corresponding to some sort or operation is red in each of the matched graphs, then the sort or operation may have changed differently in SP_2 and SP_3 . Since the result of merge must contain all the changes, but can contain only one changed copy of a sort or operation, we conclude that there may be conflict.
2. If a node corresponding to a sort or operation in SP_2 that has no corresponding sort or operation in SP_1 is red when matched against a corresponding sort or operation in SP_3 , then the sort or operation in SP_2 and SP_3 may be new, but *semantically not equivalent*. Since the result of merge must contain all the new elements in SP_2 and SP_3 , but can only contain one copy of corresponding new parts, we conclude that there may be a conflict.
3. If there is a red node in the dependency graph of SP_2 when matched against the dependency graph of SP_3 such that the node does not have any corresponding node in SP_3 , then the sort or operation corresponding to the red node is a new sort or operation in SP_2 that may depend on sorts or operations unchanged in SP_2 that may have changed in SP_3 . Since the result of the merge must contain the new parts of SP_2 along with the changed parts of SP_3 , we conclude that there may be a conflict. Similarly, the presence of unmatched red nodes in the graph of SP_3 indicates the possibility of conflict.

After we ensure that there is no possibility of conflict, the graphs of SP_1 , SP_2 , and SP_3 can be merged. The process of merging consists of the following steps:

1. Take the graph of the meet of SP_2 and SP_3 . This will represent the specification that contains the parts in SP_2 and SP_3 that have not changed from SP_1 , and the new parts in SP_2 and SP_3 that correspond (i.e., they are intended to be the same by the user) and match.
2. Collect all the non-green nodes of the dependency graph of SP_3 from its match with the graph of SP_2 . The green nodes of this matched graph are already present in the result of step 1.
3. From the result of step 2, remove all the green nodes of the dependency graph of SP_3 – except those described below – from its match with the dependency graph of SP_1 . These removed nodes are either in the graph of the meet of SP_2 and SP_3 , or they correspond to

sorts or operations that changed in SP_2 (by definition of merge, we would like to pick those changed elements from SP_2 later). We do not remove a green node that does not correspond to any node in the graph of SP_2 , because that green node corresponds to an element that has been deleted in SP_2 but left unchanged in SP_3 . By the definition of merge, we prefer to keep the unchanged part rather than delete it.

4. Add the nodes from the result of step 3 to the graph produced in step 1. These nodes represent either new elements in SP_3 , or elements that have changed from SP_1 to SP_3 , or elements that have not changed from SP_1 to SP_3 but have been deleted in SP_2 .
5. Steps 2 to 4 are repeated for SP_2 instead of SP_3 .
6. Every link in the dependency graph of SP_3 that is not in the result of step 5, and whose source and target nodes (or nodes corresponding to them in the graph of SP_2) are in the result of step 5, is added to the graph from step 5.
7. Similarly, every link in the dependency graph of SP_2 that is not in the result of step 6, and whose source and target nodes (or nodes corresponding to them in the graph of SP_3) are in the result of step 6, is added to the graph from step 6, unless its corresponding link from the graph for SP_3 has already been added in step 6.

4.4.3 Related Work

In this section, we discuss research related to our work described in this paper, and argue how this work attempts to complement and enhance the related work.

We discuss related work in three different areas as our work falls at their intersection:

1. composition operations for software development,
2. specification building operations, and
3. theories of reuse.

4.4.3.1 Composition Operations for Software Development Horwitz, Prins, and Reps [?] discuss an approach for merging non-interfering versions of programs. They describe the design of a tool based on language semantics to integrate different versions of a program. The algorithm takes as input three programs: a program A called the base program and two programs B and C that are variants of the base program. Whenever the changes made to A to produce B and C do not interfere, the algorithm produces a new program D that integrates changes in B and C with A . They have implemented this algorithm using a representation similar to dependency graphs to facilitate detection of changes and analysis of interference among changes.

Our work attempts to extend this work along two dimensions:

- With the advent of transformational techniques to transform a correct, validated specification into an efficient program (see, for example, [?]), the process of developing and formalizing software specifications has received considerable attention. Problems analogous to program version integration arise during specification building as well. When multiple developers build large specifications, the task of merging the resulting versions is critical. We have adapted the integration algorithm described by Horwitz et al. to work for the simple specification language with algebraic semantics that we have described in this paper.

- We have extended the idea of composition to include – in addition to the specification integration or merge operation – join, meet, and difference. These operations help in combining two specification modules, or in determining information common to two specification modules, or in extracting information in one specification module that is not also in the other. Our motivating example in Section ?? illustrates the usefulness of these operations.

4.4.3.2 Specification building operations In the algebraic specifications area, specification building operations have been designed and analyzed (see [?], for a survey). While most work on specification building operations has been motivated by a need to find a class of operations that are complete in some sense (for example, can a certain class of specification expressions be constructed by a given set of specification building operations, [?]), we have focused on finding operations that are necessary for certain applications such as designing reusable specifications or merging different specification versions.

In [?], the only operation that is a composition operation in our sense is the union operation; this operation is similar to our join operation. The union operation assumes that the two specifications whose union is being computed have the same signature¹⁰. We do not make this assumption; furthermore, we want to preserve properties such as consistency and sufficient-completeness of the input specifications, an issue that is not a primary focus in their work. Hence we ensure that there is no conflict between the specifications being composed before composing them.

Our composition operations can also be viewed as an extension of the concept of *evolution transformations* [?, ?, ?]. Evolution transformations are useful in evolving specifications by introducing *stereotypical changes into specifications*. But evolution transformations are not designed to assist in specification validation; they do not guarantee any properties of the resulting specifications.

4.4.3.3 Theories of reuse Methodologies for software reuse have received substantial research attention in recent years (see for example, [?]). We can view the process of reusing software as consisting of the following high-level steps [?]:

- finding components,
- understanding components,
- modifying components, and
- composing components.

In this work we have focused on the last of these steps: composition of reusable components.

Reusable components can be composed to construct other reusable components. Because components from a reuse repository are used often, it is important that they are carefully designed, and their correctness is ensured before they are deposited into the repository. Therefore, our composition operations work at a semantic level guaranteeing several important properties of the resulting composite. This helps in validating the reusable component before storing it in the repository.

Additionally, when users retrieve and compose large components, they are unaware of the interactions between these components. Hence it is essential to have automated tools that can

¹⁰Signature is simply the collection of sort and function symbols used in the specification.

compose such components and indicate any conflicts. This was also one of our main considerations in the design of composition operations.

References

- [Balzer78] Robert Balzer, Neil Goldman, and David Wile. Informality in Program Specifications. *IEEE Transactions on Software Engineering*, 4(2):94-103, March 1978.
- [Balzer83] Robert Balzer, Jr. Thomas E. Cheatham, and Cordell Green. Software Technology in the 1990's: Using a New Paradigm. *IEEE Computer*, 1983.
- [Balzer85] Robert Balzer. A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257-1268, November 1985.
- [Bhat92a] Sudin Bhat and Kanth Miriyala. A Theory for Specification Validation and Reuse. In *Submitted for Publication*, 1992.
- [Bhat92b] Sudin Bhat and Kanth Miriyala. A Theory of Composition Operations for Software Reuse. In *Working Paper*, 1992.
- [Bidoit89] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. How to make algebraic specifications more understandable: an experiment with the PLUSS specification language. *Science of Computer Programming*, 12(1989):1-38, 1989.
- [Biggerstaff89a] Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability: Concepts and Models*, volume 1. ACM Press, New York, 1989.
- [Biggerstaff89b] Ted J. Biggerstaff and Charles Richter. *Reusability Framework, Assessment, and Directions*, chapter 1, pages 1-17. ACM Press, Frontier Series, New York, 1989.
- [Boehm81] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, New Jersey, 1981.
- [Davey90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990.
- [DeBellis90] M. DeBellis. The Concept Demonstration Rapid Prototype System. In *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference*, pages 211-225, Syracuse, NY, 1990.
- [Eades89] Peter Eades and Roberto Tamassia. Algorithms for Drawing Graphs: An Annotated Bibliography. Technical Report CS-89-09, Department of Computer Science, Brown University, Providence, October 1989.
- [Feather89a] Martin S. Feather. Constructing Specifications by Combining Parallel Elaborations. *IEEE Transactions on Software Engineering*, 15(2):198-208, February 1989.
- [Feather89b] Martin S. Feather. Detecting Interference When Merging Specifications. In *Fifth International Workshop on Software Specification and Design*, pages 169-176, April 1989.
- [Fickas88a] Stephen Fickas and P. Nagarajan. Being Suspicious: Critiquing Problem Specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 19-24, 1988.

- [Fickas88b] Stephen Fickas and P. Nagarajan. Critiquing Software Specifications. *IEEE Software*, pages 37-47, November 1988.
- [Green86] Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, and Charles Rich. Report on a Knowledge-Based Software Assistant. In Charles Rich and Richard Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, pages 377-428. Morgan Kaufmann Publishers, Inc., 95 First Street, Los Altos, CA 94022, 1986.
- [Guttag78] John V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27-52, 1978.
- [Horwitz89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Noninterfering Versions of Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345-387, July 1989.
- [Inc.90] Reasoning Systems Inc. *REFINE User's Guide*. Palo Alto, CA, 1990.
- [Jalote89] Pankaj Jalote. Testing the Completeness of Specifications. *IEEE Transactions on Software Engineering*, 15(5):526-531, May 1989.
- [Johnson90] W. L. Johnson and M.S. Feather. Building an Evolution Transformation Library. In *International Conference on Software Engineering*, pages 238-248, Nice, France, 1990. IEEE Computer Society.
- [LaBatt90] Earl C. LaBatt, Jr. Analysis of Improved Many-on-Many. Technical Report RADC-TR-90-115, Rome Air Development Center, April 1990.
- [Miriylala92] Kanth Miriyala, Scot W. Hornick, and Roberto Tamassia. An Incremental Approach to Aesthetic Graph Layout. 1992.
- [Musser80] David Musser. Abstract Data Type Specification in the AFFIRM System. *IEEE Transactions on Software Engineering*, SE-6(1):24-32, January 1980.
- [Reubenstein91] Howard B. Reubenstein and Richard C. Waters. The Requirements Apprentice: Automated Assistance for Requirements Acquisition. *IEEE Transactions on Software Engineering*, January 1991.
- [Smith90] D. R. Smith. KIDS - A Semi-Automatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024-1043, September 1990.
- [Srinivas90] Yellamraju V. Srinivas. Algebraic Specification: Syntax, Semantics, Structure. Technical Report TR 90-15, University of California, Irvine, June 1990.
- [Wing88] Jeannette M. Wing. A Study of 12 Specifications of the Library Problem. *IEEE Software*, pages 66-76, July 1988.
- [Wirsing90] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter Formal Models and Semantics, pages 675-788. MIT Press/Elsevier, 1990.

Section 5: A Software Development Process Model for the KBSA Concept Demonstration System

This section describes, at several different levels of description, one likely KBSA-based software development process. It does not attempt to present the definitive KBSA software development process model, but rather a convincingly detailed example of one possible model. As other KBSA researchers have noted, there will probably never be a single KBSA process model — one of KBSA's greatest strengths is its potential ability to support alternative process models (Jullig, 1989). However, in order to understand and discuss the issues involved in developing process models for KBSA, we believe it is essential to develop at least one detailed example of a potential model.

Section 5.1 will present an overview of the process model, and section 5.2 will discuss how the Concept Demo represents elements of the process model (e.g., tasks, transformations, and kb-module states) internally.

5.1. High-Level Vision of the Concept Demonstration System Process Model

This section describes a high-level vision of a process model for software development based on a Knowledge Based Software Assistant (KBSA). Further, it relates that process model to the functionality present in the final delivery version of the KBSA Concept Demonstration System.

5.1.1. Overview of the High-Level Process Model

Distinctions Between the KBSA and Waterfall Models. The KBSA-oriented software development process model discussed below presents its users with three major advantages relative to the conventional Waterfall development practices:

- Formal representation and manipulation of the target system specification without committing to a specific implementation, enabling strong validation of via simulation, prototyping, and other evaluation techniques and system-level (rather than program-level) optimization for better performance;
- Tightly integrated, rapidly cyclical, incremental development, enhancing continuity between successive development artifact states and enabling improved traceability and replay of the development process; and
- Sophisticated and integrated process modeling capabilities, allowing control of development without forcing a waterfall-like, lock-step progression on the set of development artifacts.

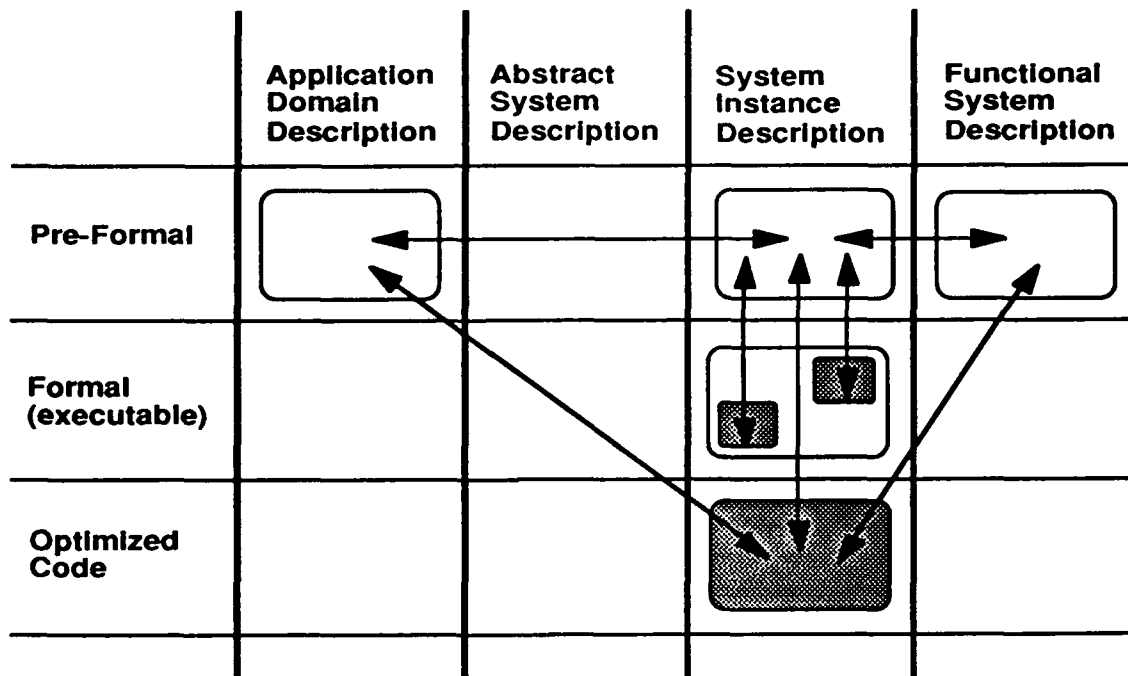


Figure 5-1: How Conventional Models Treat Development Information

As shown in Figure 5-1, conventional, waterfall development typically captures three types of information defining the system to be built: an application domain description (e.g., an entity-relation diagram), a system instance description (e.g., a Yourdan structure chart), and a functional system description (e.g., a functional specification). This information is captured, organized, and integrated at an pre-formal level, and the system instance description is then elaborated into optimized code. In some cases, parts of the system instance description are elaborated into formal specifications via prototyping or simulation models. These are (sometimes) used to validate the pre-formal instance description, prior to its implementation and optimization in code. Note that, in most cases, these executable partial specifications (i.e., the prototypes and simulations) do not feed forward into the actual code development process.

Figure 5-2 shows that KBSA handles this information in a significantly different way. First, it makes far greater use of formal (i.e., executable) specifications. This enables incremental development and increases the power of the validation process. Second, KBSA makes explicit use of a new category of information, the abstract system description. The abstract system description is an executable specification, synthesizing the desired system behavior and the key application domain models, prior to any major commitment to implementation techniques. The KBSA-supported developer defers optimization issues until the abstract system model has been elaborated and validated.

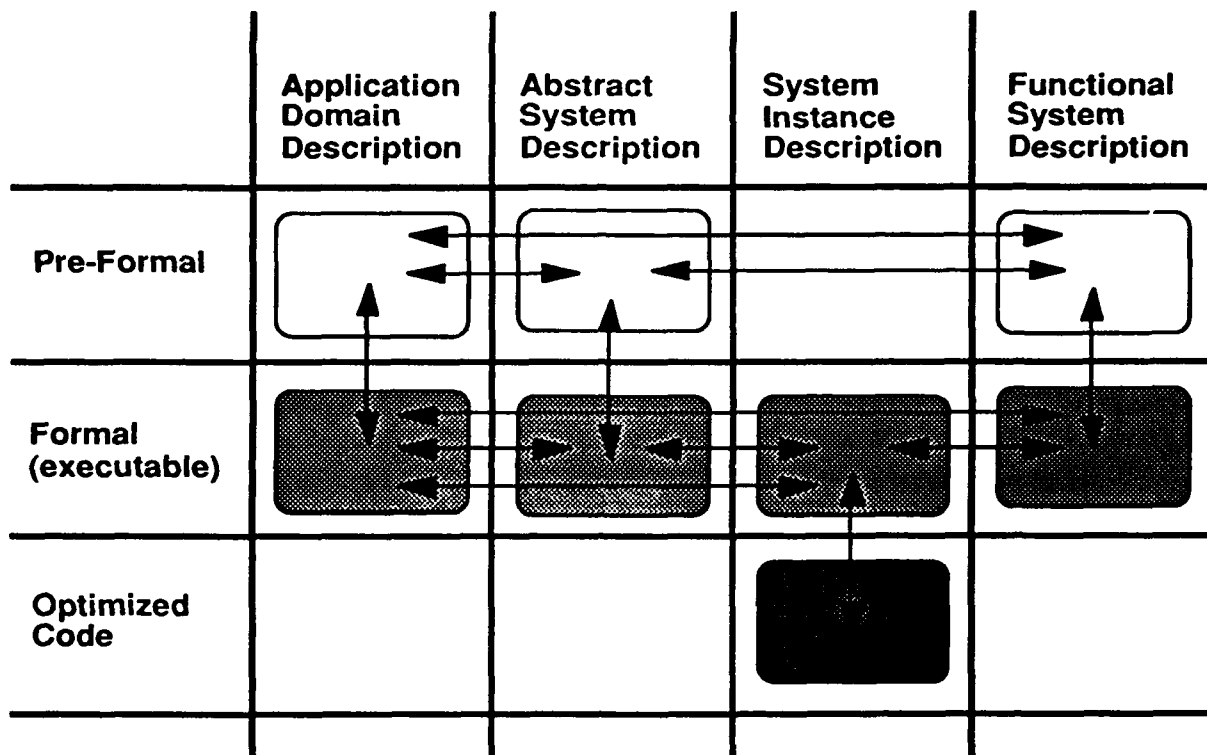


Figure 5-2: How KBSA Treats Development Information

Structure of the Section. Section 5.1.2 describes briefly the previous work from which our model is derived, and presents our process model at an overview level, depicting the complete spectrum of software development and evolution. Section 5.1.3 then presents more detailed discussions of the model's treatment of requirements in their pre-formal state. Section 5.1.4 discusses how they are then formalized as implementation-independent, declarative requirements. Within the Concept Demonstration project, these processes received the bulk of our effort and attention, and are thus more fully described than is specification implementation in section 5.1.5. After a brief descriptions of the relationship between this model and Boehm's Spiral Model in section 5.1.6, section 5.1.7 presents a summary of section 5.1.

5.1.2. Presentation of the Process Model

Relationship to Previous Research. Our Process Model integrates research from three different areas: (1) knowledge-based software engineering, (2) Artificial Intelligence (AI) planning and its application to software development, and (3) studies of the cognitive, social, and organizational processes employed in real-world software development.

The transformational development paradigm (Balzer, 1985) provides the foundation for our process model. The KBSA research program has been created around an elaborated vision of that paradigm (Green et al, 1983), and many of the atomic tasks represented in our model are transformations developed in earlier KBSA program work (Johnson and Feather, 1990; Goldberg et al, 1989; Smith, 1990). Similarly, the task formalism that we present here is an extension of that developed in an earlier KBSA research effort, the Project Management Assistant (Daum and Jullig, 1990).

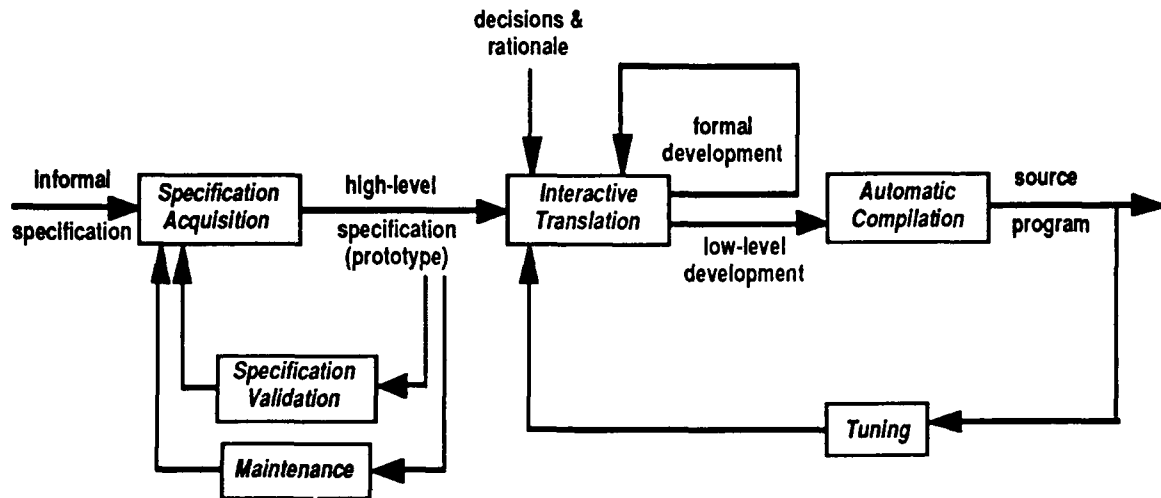


Figure 5-3: Balzer's Extended Automatic Programming Paradigm

Intelligent software development environments, such as MARVEL (Kaiser et al, 1988) and Grapple (Huff, 1989), have defined software development task and object formalisms that AI planning techniques can operate on. For example, backward chaining can be used for plan generation. This will identify the sequence of tasks that will move an object (e.g., a specification) from its current state to a desired state. Our model applies this strategy, but at a finer granularity, planning for the development of individual objects as well as for components consisting of multiple objects.

Work done at MCC by Guindon et al (1987) and Curtis (1990) illustrates the importance of enabling the developer to move between the highly conceptual and implementation-specific levels of analysis. Their results suggest that the rich process description information in process programming models (Osterweil, 1987) should be combined with a more flexible, adaptable planning approach such as that discussed by Huff (1988). This gives the human developer control of the development process through a mixed initiative interface, while providing him with powerful development capabilities.

Perry (1988) discusses the importance of issues of scale and number of developers, which many process modeling formalisms fail to address. Barghouti and Kaiser (1990) have begun to address some of these issues in their Multi-User MARVEL model. Mi and Scacchi (1990) identify several standard techniques which can be used to coordinate the work of multiple developers when resource conflicts occur. Their formalism has been used to analyze and simulate methodologies for large development teams. While our work to date has focused on guidance for the individual developer, our formalism for tasks and states is almost identical to theirs. Therefore, we believe that we can provide similar capabilities by integrating their process representation meta-model with ours.

Section 5.2 below will describe how our work combines AI planning with a knowledge-based implementation of the transformation-based development paradigm. As noted by Huff (1988), this provides the planning mechanism with more detailed, rigorous, and complete knowledge about the current state of the specification. Similarly, the development operations are described in greater detail, so planning can actually produce a list of specific, detailed development steps — many of them executable — that will achieve the project objectives. To ensure that the human directs the software development process, we provide a mixed-initiative interface which gives the user both control and responsibility.

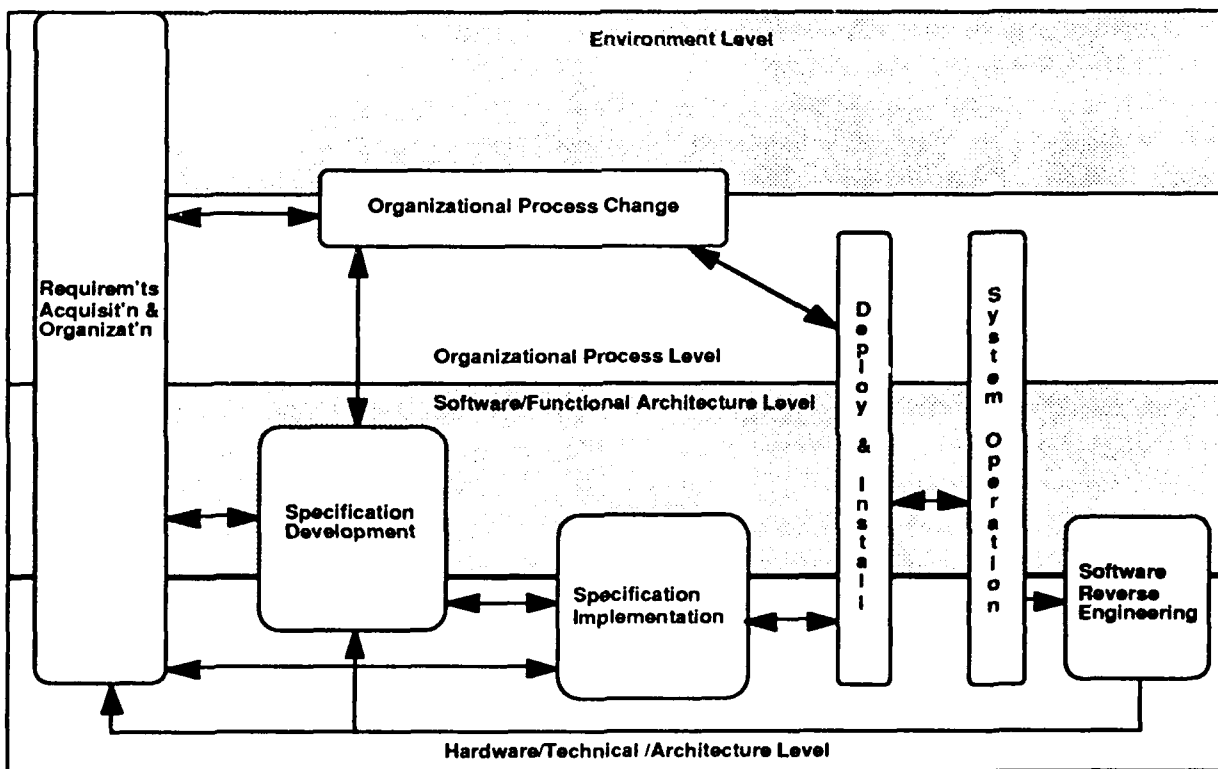


Figure 5-4: High-Level Process Model

Overview of a KBSA-Oriented Process Model. This model is a direct descendent of Balzer's. It extends his work by explicitly considering (1) the acquisition and organization of informally expressed requirements, (2) the communication and contention issues of programming in the large, (3) the reuse of specification elements, and (4) the inclusion of reengineering as a means of developing formal specifications. The process model we present here is discussed primarily in terms of the forward engineering process, but it has strong assumptions concerning the importance of reverse engineering and the evolution of deployed systems. Figure 5-4 represents a high-level view of the development process. The basic concept illustrated in the Figure is that KBSA-based development progresses through four major activities:

1. Acquisition: capture of requirements, in text and/or graphic form, followed by their disambiguation and organization into a well-formed structure of requirements in preparation for their formal expression;
2. Specification development: formalization of requirements and their incremental development via evolution transformations (cf. Johnson and Harris, 1990) to a detailed but still implementation-independent state;
3. Specification implementation: application of meaning-preserving transformations to optimize these formal requirements into specifications (implementation-specific statements suitable for compilation and operational deployment); and
4. Deployment and installation of the production-quality system, followed by its ongoing operation, and evolution.

The background shading of the Figure is intended to convey that different types of knowledge are required for each of the activities in the process model.

Knowledge of the Environment: The environment refers to knowledge external to the organization itself, such as Government regulations, relevant properties of physical laws, and standards for financial accounting. The organization typically has little control over these, but their implications for the functionality of the application must be recognized early in the development process. For example, a system that controls a petroleum refinery must include robust models of the relevant chemical reactions, just as an asset management system must use accepted depreciation calculation methods.

Knowledge of Organizational Processes: Organizational process knowledge describes practices within the organization, such as the pricing of products, the assignment of business responsibilities to personnel, or the identification of objects on radar as hostile or friendly. In most cases, the application software to be built (or re-engineered) is intended to support the performance of these processes. For example, the U.S. Federal Aviation Administration's current

practice is to divide air traffic control responsibilities between different groups of controllers responsible for ground control, departure and approach, and flight en-route. The requirements need to capture this information such that the appropriate data can be displayed effectively to each of these controllers.

Knowledge of Software/Functional Architectures: This level includes the knowledge that the application system is expected to contain, and describes the behavior it is expected to exhibit. For example, an ATC system is expected to contain representations of (or "know about") aircraft, pilots, separation rules, airports, and controllers. Its behavior will typically include the ability to display a representation of an airspace sector and to maintain a common database of flight information in order to support air traffic controllers. Knowledge at this level also includes non-functional requirements and software development techniques.

Knowledge of Hardware/Technical Architectures: At this level we describe the target system's intended execution environment in terms of hardware and system software. In some cases, the constraints imposed by the technical environment will need to be considered in the formalization of specifications or even earlier in the description of requirements. For example, if the application is constrained to use IBM-3270 technology for its interface, most user interaction will have to be handled through character-based input and output. Alternatively, the knowledge that the application system is expected to operate in an environment offering more powerful graphical user interface capabilities enables the developer to design a completely different (graphical) interface.

The organizational process change activity shown reflects our recognition that new application software will generally imply adjustments to the organization's operating procedures, training programs, hiring practices, and possibly other areas as well. In addition, plans for conversion and installation should be developed prior to the deployment step. We will not address this activity further here, but recognize that it will require additional consideration before KBSA technology can be successfully transferred to actual production usage.

Software reverse engineering abstracts design information from the code and documentation of existing systems, and reimplements the design in a more maintainable form. Reverse engineering allows organizations to move from a conventional to a KBSA-based environment. Today's organizations have mission-critical software that is very difficult and expensive to maintain. For these organizations to realize the full benefits of KBSA, they will need to reverse engineer their legacy systems into KBSA, because:

1. Up to 80% of software development effort goes to maintain existing software rather than implementing new software (Martin and McClure 1983).

2. Even completely new systems developed with KBSA will have to be integrated with existing systems. This integration will be far more difficult if these systems remain based in static, inflexible, non-KBSA technologies.

Unfortunately, although the importance of reverse engineering to KBSA been articulated (Kozaczynski and Ning, 1989; Kotik and Markosian, 1989; Newcomb, 1989), much work remains to be done on reverse engineering into KBSA.

5.1.3. Acquisition and Organization of Pre-formal Requirements

Acquisition and organization of informally expressed requirements logically precedes the development of formal requirements. This involves the identification and organization of relevant information, application domain descriptions expressed in natural language. We use the term *pre-formal* to describe requirements in this state, where the requirement is expressed informally, but the informal expression is a component of a formal object (a hypertext requirement object) with formally expressed relationships to other hypertext objects. Requirements can be differentiated into three types of information: application domain descriptions, user expectations of system behavior, and the designer's preliminary concept of the evolving system (Johnson and Harris, 1990, Johnson and Feather, 1990). Each requirement can be linked with others to form kb-modules. We envision that the activities below would typically be used to capture, organize, and refine requirements. This is not a fixed sequence of activities to be applied mechanically to every requirement, but rather a set of operations to be applied in various orders as the nature of the case requires.

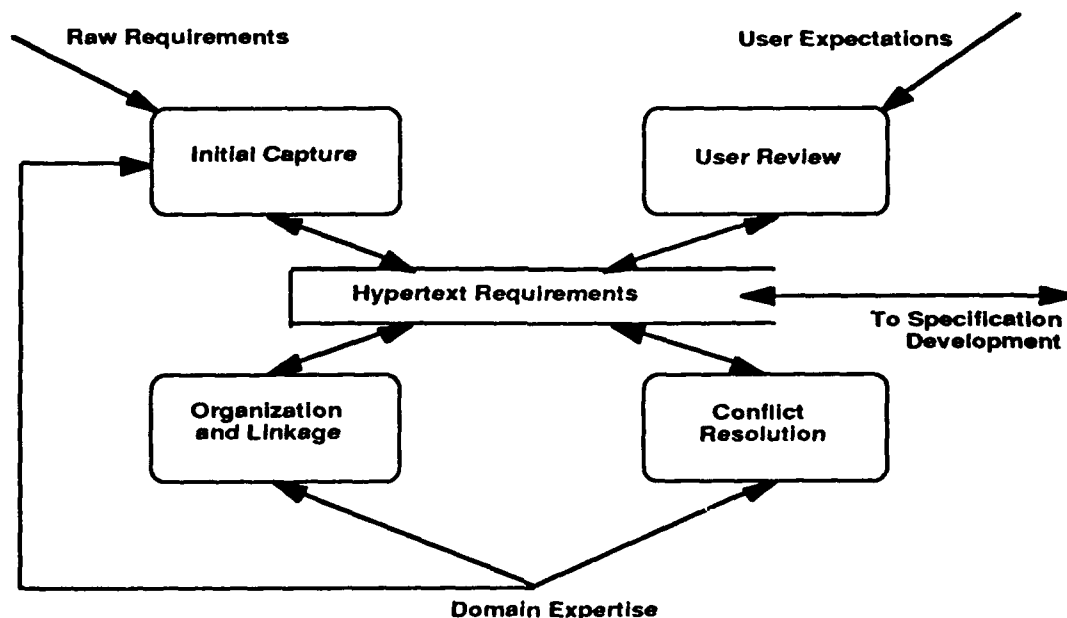


Figure 5-5: Requirements Acquisition and Organization

- Initial capture of an atomic requirement element. For example, "The minimum vertical aircraft separation should be 1,000 ft up to flight level 290." We would also capture information describing the source or owner of the requirement (in this case, FAA document 7110.65, page 5-5-1).
- Objective (automated) clustering by ownership and commonly occurring terms. For example, the sentence above might be automatically clustered with other declarations concerning "minimum ... aircraft separation."
- Subjective (user-assisted) clustering on the basis of related concepts. For example, we might link aircraft separation rules with a domain model of the jet-route network, since the capacity of the jet-routes can only be determined with reference to the minimum separation standards in effect.
- Identification and resolution of incompleteness and incorrectness. In terms of the above example, we might note that flight levels are units of altitude measurement corresponding to 100 feet (i.e., flight level 290 is 29,000 feet).
- Resolution of conflicts/contradictions internal to kb-modules. Suppose, for example, that one requirement states that "All indicators of aircraft, airspace, and radar will appear on the controller's display at all times" while another states that "To improve usability, no more than 50 distinct items will be displayed at any given time." These are potentially in conflict, should the airspace region displayed contain more than 50 items. The application designer might resolve this by requiring that the system allow the "graying out" of some of the less important items when the display includes more than 50 items, or by providing several types of display filtering options.
- Review and approval of individual kb-modules. For instance, after the appropriate users have agreed that the pre-formal expression of the separation requirement has been expressed accurately, the developer can begin to formalize that requirement.
- Resolution of conflicts/contradictions between kb-modules. For example, the separation kb-module and the human-computer interaction standards kb-module might contain contradictory expectations regarding the interface design. These will need to be addressed (at least to the extent of identifying tradeoffs between different goals of the target system) before the modeling and evaluation of the conflict in the specification development process can take place.

Iterative review of requirements will occur even at the pre-formal requirements state, as users examine the relationships stated in the preliminary description of the domain and system functionality. Once they agree that it represents their intentions and domain knowledge accurately, it will be placed in a modification controlled state. Automated analyses will be conducted to ensure that each kb-

module is well-formed. For instance, this could involve determination that each question attached to a requirement has a response attached to it.

The KBSA Concept Demonstration System supports requirements acquisition and organization via a hypertext capability (Johnson et al, 1990 and DeBellis, 1990) and the use of a presentation architecture (Harris and Czuchry, 1988). Automated text input, display of the structure of the pre-formal hypertext requirements plex, and requirements browsing capabilities are also available.

5.1.4. Development of Formally Expressed Requirements

We use the term *formally expressed requirement* (or *formal requirement*) to denote the next state of development in the KBSE process model. Our use of formal implies simply that the requirement is expressed in a machine-intelligible language — that the formal requirement can be compiled. Our choice of the term requirement signifies the declarative nature of the system description. A formal requirement that describes a process, for example, will identify its preconditions and postconditions, but will not include a procedural description of an algorithm. The algorithm description will be added in the specification implementation stage.

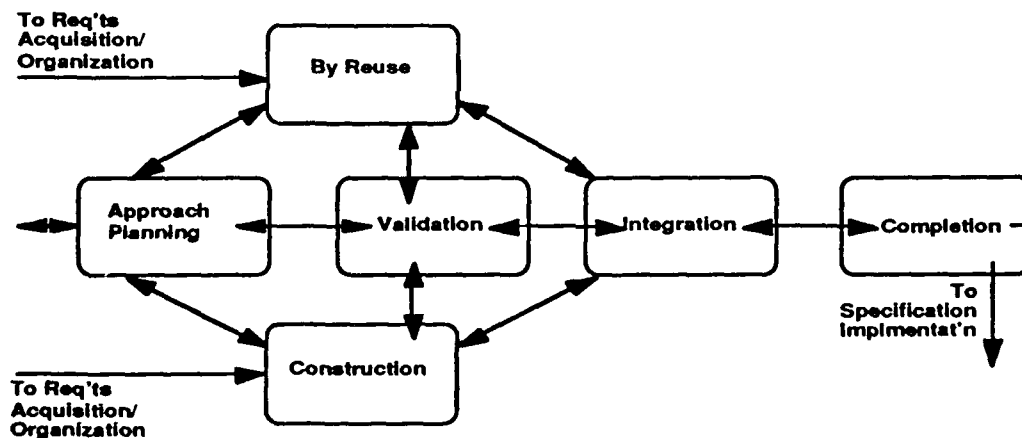


Figure 5-6: Formalization of Requirements

The iterative process of formally expressing the pre-formal requirements can begin at any time. Generally, however, it will follow their acquisition, organization, and review. As shown in Figure 5-6, the formal expression activity allows two main approaches: reuse and construction. After one (or possibly both) of these approaches has been selected, the formal requirement is elaborated and extended. Periodically, it will be validated by the owners of its associated pre-formal requirements. Typically, the formal expression will cycle through several elaboration, validation, and revision processes, as inconsistencies, incompleteness, and inaccuracies are uncovered and corrected. These iterations will also include modifications to the pre-formal requirements to ensure that they

stay in sync with their formal expressions and to maintain traceability. Once the requirements been formalized, they will be combined with other modules and any inconsistencies will be rectified. Finally, any remaining incompleteness (such as unintentional non-determinism, omniscience, and omnipotence¹) will be corrected in a completion task (Johnson, 1990).

A typical formal element will begin as an pre-formal requirement. Using the incremental construction approach, the developer will use a combination of the following operations to formalize and elaborate the pre-formal elements of a kb-module:

- Basic Knowledge Base operations will create formal objects corresponding the pre-formal requirement.
- Evolution transformations will elaborate these objects, replace their default values with appropriate context-specific ones, and capture their more complex relationships (e.g., define rules and relationships between objects).

To validate, revise, organize, and complete the formal requirements, the developer will use an incremental process combining features of prototyping with elements of the Spiral Model (see Boehm, 1988). This incremental process will iterate through the following operations:

- Automated analysis will identify problems of completeness and consistency within each formal requirements kb-module, and suggest evolution transformations that can be used to correct these problems.
- Users and developers will validate the formal requirements via critiquing paraphrases and the behavior of prototypes and/or simulations.
- Developers will apply evolution transformations to revise the formal requirements until the behavior of the prototypes and simulations is considered correct, and their associated pre-formal requirements are also updated.
- Developers can then apply powerful kb-module transformations, such as union, intersection, and difference operations (described in section 4 above), to integrate the individual kb-modules, removing inconsistencies and

¹Non-determinism occurs, for example, when a specification defines a set of distinct options without indicating how selection among them is to be made (e.g., a constraint stating that there exists some controller for each controlled aircraft, without stating how a controller is to be assigned to an aircraft). An omniscient specification assumes that it has access to any data it requires, ignoring the question "Where did that data come from?" An omnipotent specification is one which assumes that a change made in its knowledge base (e.g., a change in the value of aircraft-heading) will automatically cause the corresponding change in the environment (the physical aircraft will actually change its heading).

duplication between them (while validating that their behavior continues unchanged).

- Automated analysis, as well as review by users and developers, will identify any remaining non-determinism, omnipotence, or omniscience, and developers will apply evolution transformations to revise the kb-module.

At this point, the developers will have produced a formally expressed, declarative statement of the functionality required of the target application system. Additional types of information, such as the target application's intended execution environment and its non-functional requirements (e.g., response time, throughput, capacity, etc) will also need to be specified for viable large-scale development. These non-functional requirements would also be initially captured and organized as pre-formal hypertext, and later expressed as formal requirements.

In the KBSA Concept Demonstration System, basic knowledge-base manipulation commands are used to create the initial formal objects. These are then extended and refined via evolution transformations similar to the high-level editing commands developed in the Knowledge-Based Specification Assistant (Johnson, 1987) and the ARIES project (Johnson and Feather, 1989). Powerful evaluation techniques such as paraphrasing, prototyping, and simulation complement the consistency and completeness analyses performed automatically by the KBSA Concept Demonstration System. The Concept Demo uses a presentation architecture (similar to that described in Harris and Czuchry, 1988) to maintain consistency between different presentations.

5.1.5. Specification Implementation

The formal requirements passed forward to specification implementation will be declarative constructs, such as demons, constraints, and class hierarchies with inheritance and default values. The default method of compilation for some constructs may be very inefficient and none will be optimal. Specification implementation will transform these constructs into more efficient, lower level constructs via facilities such as data structure selection, unfolding of constraints, state saving (to handle historical reference), finite differencing, and loop fusion. We will use the term *formal specification* to denote a system description model in this state.

These techniques have been examined in the research performed at the Kestrel Institute in the KIDS environment (i.e., Smith, 1990) as well as at other research centers (cf. Balzer, 1985). The amount of effort required in this stage will depend upon a number of factors:

- Complexity of the problem. For example, implementation of a specification for a real time scheduling system will require all possible optimization techniques.
- Capabilities of the run-time environment. For example, in some environments it will not be possible to directly compile a specification with global knowledge base or constraint constructs.
- Non-functional requirements. For instance, response time, reliability, and other non-functional constraints to guide the specification implementation process.
- Sophistication of the compiler. Some of the transformations which are now user guided (such as those demonstrated in the KIDS environment) may eventually be automated, as compiler technology continues to become more powerful.

In contrast to the transformations used in specification development, those used here will be meaning preserving — ideally, this will have been proven to be true for them. They will improve run-time characteristics of the target system (e.g., performance). Most of the previous research on these transformations has emphasized processing optimizations, which are very important, especially for real-time systems. But other types of implementation transformations will also require attention before KBSA will be usable in industrial environments. These include data-oriented optimizations and system architecture selections. Systems which deal with very large amounts of data require different types of optimizations than do process intensive systems. These optimizations must consider low-level details about the structure and access languages of the system's various data-bases. System architecture factors play an essential part in the implementation decisions which real programmers make. KBSA needs to represent and use such knowledge to integrate different architectural environments efficiently and seamlessly.

Following Concept Demo's goal of leveraging existing KBSA software, we have integrated with the KBSA Development Assistant (Smith, 1991) to provide specification implementation functionality. As noted above, this work has dealt mainly with process optimizations.

5.1.6 A Spiral View of the Concept Demo Process Model

Another view of this process model relates it to the Spiral Model described by Boehm (1988). This model identifies a set of discrete development cycles, each consisting of a planning, execution, and evaluation phase. Evaluation of the each cycle's outcome guides the planning for the next cycle. Each cycle reduces project risk by concentrating development effort on the most critical issues facing the project. For example, suppose the application's functional requirements are

familiar to the developers but a new type of interface technology is desired (e.g., digital video interactive). In this case, the spiral model would suggest that a feasibility study of the new interface technology should precede a detailed requirements analysis.

The following sequence of topics could represent a spiral model development process in a KBSA-based environment:

- Capture basic requirements and system interfaces
- Elaborate pre-formal requirements
(functional, non-functional, and architecture in parallel)
- Formalize pre-formal requirements
(functional, non-functional, and architecture in parallel)
- Investigate reuse possibilities
- Elaborate formal requirements
(functional, non-functional, and architecture in parallel)
- Merge functional requirements and architecture into system model
- Model and evaluate performance and reliability (revise system model)
- Optimize functional specification using meaning-preserving transformations
- Generate code from optimized specification
- Install application and convert existing data and procedures
- Operate and evolve application

5.1.7 Summary

This section has presented a high-level overview of one possible process model for software development in a KBSA environment. The model's major activities are organized around the degree of formalization and elaboration of the software development artifact. They need not proceed in lockstep, but rather provide the developer with strongly integrated facilities enabling rapid cycles of software development, validation, elaboration, and ongoing enhancement. The model recognizes that support is needed for the acquisition and organization of pre-formal requirements information as well as for the creation and extension of formally expressed specifications, and extends an earlier model to do so. It recognizes the importance of interfacing with existing systems and the recovery of procedural knowledge embedded within them, and provides these capabilities by inclusion of Software Reverse Engineering as a top-level activity.

5.2. Plan-Based Guidance for Knowledge-Based Software Engineering

This section describes a Process Model formalism based on Artificial Intelligence (AI) planning techniques which has been developed for the Knowledge-Based Software Assistant (KBSA) Concept Demonstration System. This formalism integrates previous work using planning representations to model software development (e.g., Kaiser et al, 1988, and Huff, 1989) and work on knowledge-based software development, such as the KBSA Specification Assistant (Johnson and Feather, 1990) and the Kestrel Interactive Development System (Smith, 1990). A central feature of these environments is their use of powerful transformations to support the development process (Green et al, 1983). A transformation is a formally expressed, machine executable operation that changes the state of a program or formal specification. Transformations change programs in controlled, systematic ways to introduce or enhance desired features. For example, they can improve execution efficiency or elaborate descriptions of the system's functionality or context.

5.2.1. Introduction

By integrating an AI planning formalism with a knowledge-based software engineering environment, the Knowledge-Based Software Assistant Concept Demonstration System exemplifies significant advantages over conventional development support. These include (1) stronger guidance in the selection of development operations; (2) better development process traceability; (3) more powerful development support; (4) finer-grained planning capability; and (5) mixed-initiative development. This section will describe the planning formalisms used to accomplish this integration, and present an example illustrating these benefits.

By superimposing a plan-based formalism on a knowledge-based environment we achieve the following benefits:

- Stronger guidance in transformation selection. Even experienced users of knowledge-based development environments can have difficulty selecting the appropriate transformation sequence to achieve a development goal. The planning formalism helps developers select goals and construct transformation sequences to achieve them.
- Better development process traceability. Because the planning formalism provides access to the rationale for each transformation, we can record not only the sequence of transformations applied, but also information relating the transformation's local goal to the higher-level software development process goals.

By extending plan-based formalisms to model knowledge-based development environments, we achieve a complementary set of advantages:

- Higher-level development operations. Knowledge-based development increases development productivity because — using transformations on software development objects (e.g., objects and relations in the application domain) rather than operations on edit-level objects (e.g., characters, lines, or regions in an Emacs text file) — it accomplishes far more work in each atomic action of the developer.
- Finer-grained planning capability. Transformations with strong semantics replace editing tasks with very weak semantics as the leaf nodes in the plan structure. In conventional software development environments, most development tasks employ low-intelligence tools such as text editors. For example, adding a new entity to an E/R diagram (see the example presented on pages 7-11 below) and propagating this change through other affected design documents (e.g., data-flow diagrams) require several manual editing steps for each diagram. Through the use of transformations, this activity can be executed in a single atomic step.
- Mixed-initiative development. Rather than force developers to change their work style to fit the planning system's model, we follow Huff (1988) in employing a mixed-initiative work style. As users execute tasks, the planner can recognize plans "bottom-up" and relate them to the process model. The stronger semantics of the transformation-based formalism enhance the planner's ability to do this. When stronger guidance is desired (for example, to support novice users) the Concept Demo can employ a top-down planning approach, in which the user selects a high-level task and allows the system to guide her to its completion, step by step.

This section is structured as follows. Section 5.2.2 will explain the knowledge representation formalism used in the Concept Demo to provide the advantages described above, and section 5.2.3 will then present a detailed example showing how the formalism supports them. Section 5.2.4 then concludes with a discussion relating the formalism to other work on process models and a brief summary of possible directions for future work.

5.2.2. Representation of Process Knowledge

As described above, the Concept Demo model of software development is derived from that of Balzer (1985) and modeled more generally on the KBSA approach (Green et al, 1983, and Sasso and DeBellis, 1990). Software development in the Concept Demo begins by capturing informally expressed requirements. These are next formalized as declarative requirements (e.g., invariants, functions, demons, or methods expressed formally). Later, implementations are derived for these declarative requirements using either algorithm design techniques (Smith, 1990) or transformations (Johnson and Feather, 1990). Optimization is then performed on the specification before code is

derived. This is done using data structure selection and algorithm optimization transformations. For example, a data structure defined at the declarative level as a set might be transformed into either an array or a list, depending on whether or not its maximum size is known. Loop fusion, which merges two iterations over a set into a single iteration (cf. Goldberg et al, 1989) exemplifies an algorithm optimization.

The Concept Demo uses a hierarchical planning representation to model this process. The two primary constructs in this representation, states and tasks, are organized in hierarchical state-transition diagrams. The transitions represent tasks which change the states of development objects (requirements, specifications, code) in the developer's workspace. A state description is a predicate used to test the status of one or more of these objects. Since development objects and project management information are represented in an integrated knowledge-base, it is possible to formalize descriptions of important states in the progress of system development. This enables the Concept Demo to evaluate the status of the developer's workspace automatically, and to identify transformations which he can apply to achieve his development goals.

Tasks are hierarchically related: each task is either atomic or has one or more sub-tasks. Tasks may be executable, non-executable, or mixed-initiative in nature. An executable task can be carried out with little or no human guidance. Examples of executable tasks include transformations and automatic analysis routines. To carry out an executable task, the developer needs only to select the task and (perhaps) respond to system prompts for various parameters. Examples of non-executable tasks would be gathering and modifying informal requirements and reviewing a prototype generated from a specification with an end-user.

As an example of this formalism, consider the Splice-Communicator task. This is an atomic, executable task which can be performed through invocation of the "Splice-Communicator" transformation. The transformation's input is a group of development objects with an associated marker indicating an illegal slot access. Its preconditions would determine that the object-group is checked out to the developer for modification and that the illegal slot reference occurs in the object-group. Its primary output is a new version, elaborated to include a process defined to access the slot legally. Splice-Communicator is atomic, so it will not have a subtask sequence. Its supertask might be, for example, Resolve-Illegal-Slot-Access. A (simplified) specification of the splice-communicator task is shown in the Appendix. This specification is expressed in Extended Refine Specification Language (ERSLa), as described in DeBellis (1990).

The Representation of Development Objects and Their States

Splice-Communicator, like many other transformations, operates on groups of objects, rather than single objects. The Concept Demo uses a structure called a

knowledge-base-module (kb-module) to organize the objects in the developer's workspace into these groups. A kb-module is special class of object, which provides encapsulation at a higher-level of granularity than that provided by existing object-oriented representations. In traditional object-oriented representations, an object can hide its behaviors and states by declaring its methods and slots to be public or private. A kb-module provides encapsulation at the object (as opposed to the method or slot) level. Objects are associated with kb-modules and — unless explicitly declared as exported — the objects in one kb-module are hidden from those in other kb-modules.

As shown in Figure 5-7, each kb-module in the KBSA Concept Demo goes through four high-level states:

1. Pre-formal KB-Module: containing some pre-formal objects (hypertext requirements or informal diagrams) that have not been linked to formal representations.
2. Formal Requirement KB-Module: containing pre-formal objects which have all been linked to formal declaration objects, some of which are not yet associated with formal implementation objects.
3. Formal Specification KB-Module: containing a formal implementation object for each formal declaration object, and a formal declaration object for each pre-formal object.
4. Optimized KB-Module: containing a formal implementation object for each formal declaration object, a formal declaration object for each pre-formal object, and associating optimized implementation objects with the formal implementations wherever appropriate.

Two of the transitions in the diagram are bi-directional: the one between pre-formal and formal requirements, and that between formal requirements and formal specifications. The bi-directionality indicates that an iterative process is often involved in these transitions. For example, in adding implementation-level detail to a formal specification, one may uncover the need to include additional requirements.

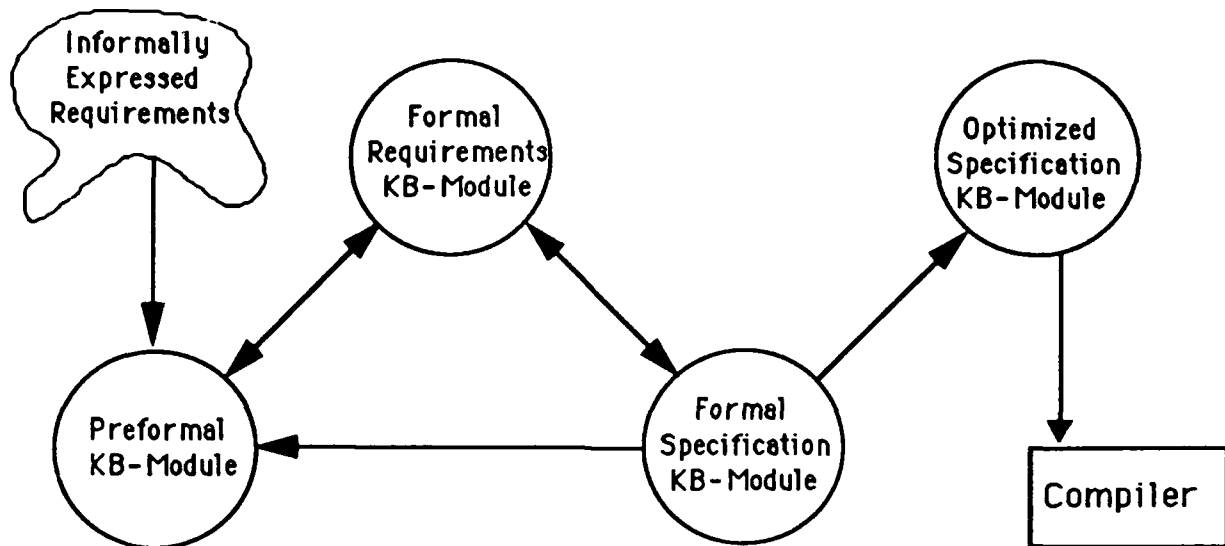


Figure 5-7: Highest-Level State Model

We formally define the four high-level kb-module states enumerated above as predicates, so the Concept Demo can monitor the status of a kb-module at any time. In Appendix 5.2-A, we show these predicates expressed formally in ERSLa.

Modeling Three Levels of Software Process Activity

The Concept Demo models development activity at three levels: the kb-module, the developer's agenda, and the project. At the kb-module level, the developer manipulates a set of related knowledge-base components organized into a kb-module. The kb-module level corresponds to "programming in the small." At the agenda level, activity focuses on the identification and prioritization of various tasks assigned to an individual developer. Models at the agenda level can be thought of as corresponding to a "programming by the one" paradigm. The project level integrates the responsibilities and outputs of many developers, working either as individuals or in teams. This level encompasses both "programming in the large" and "programming in the many."

Interactions between the kb-module and the agenda levels occur when an activity at the kb-module level, such as elaboration of a specification, introduces items into or removes items from some agenda. After a change in the workspace, the agenda mechanism is automatically invoked to create new agenda-items appropriate to the state and the process model and remove those referring to goals that have been satisfied.

The Concept Demo's agenda mechanism can propose specific executable tasks that are likely to resolve issues currently on the agenda based on its knowledge of states, operations, and goals. Issues on the agenda are represented as goals

(which are predicates). Tasks have preconditions and postconditions (which are also predicates). In the simplest case, a goal might match the postcondition of a certain task and its precondition might already be satisfied, allowing the task to be posted to the agenda as a proposed solution for that issue. In more complex cases, the task's precondition may not be satisfied, so it becomes the active goal. This backward chaining may continue until the system finds a sequence of operations that transforms the current state into the desired goal state (or until it determines that no such sequence can be automatically constructed for the goal).

Interactions between the agenda and project levels occur when an activity at the project level introduces changes to a developer's agenda, or when a change to an agenda has implications for the overall project. For example, removal of an agenda-item may imply completion of a task which allows logically subsequent tasks to begin. Agenda-items are only removed when one of three conditions is true:

- the executable task associated with the item successfully executes;
- the user marks an agenda-item as completed, provides a justification, and the Concept Demo verifies that the item's postconditions have been satisfied; or
- external changes to the knowledge-base make the agenda-item irrelevant (for example, if an item identifies a syntactic problem in a specification statement, and the statement is deleted, the item will automatically be removed).

In each case it is possible to record automatically the date and time that the agenda-item representing the final postcondition of a task was completed, as well as the identity of the agent who completed it. Similarly, monitoring the status of the set of agendas enables determination of the overall project status and the identification of potential difficulties in time for management to take action.

5.2.3 An Extended Example

We now present a detailed example which illustrates the advantages of our approach. Our example focuses on an Air Traffic Control (ATC) project kb-module with two sub-modules. The atc-domain sub-module models the aircraft domain. It contains classes such as aircraft, pilots, flight-plans, controllers, etc. The atc-system sub-module is a specification for a system to monitor and control aircraft. Figure 5-8 shows three sub-states of the formal specification state (gray area) along with the preceding formal requirements state:

- A specification kb-module is considered unresolved if it has unresolved issues attached to one or more of its formal objects. We have incorporated the model used in the graphical Issue-Based Information System (Conklin and Begeman, 1988) to represent issues and resolutions relating to systems.

During development, users, managers, domain experts, and developers indicate potential problems with a kb-module by attaching gIBIS objects to the offending objects.

- A kb-module is resolved if all known issues have been resolved, but the developer has not yet executed the validation task. The developer may be awaiting the readiness of another kb-module, perhaps containing its test cases. Alternatively, he may plan to elaborate some elements of this kb-module further, but may not yet be ready to do that.

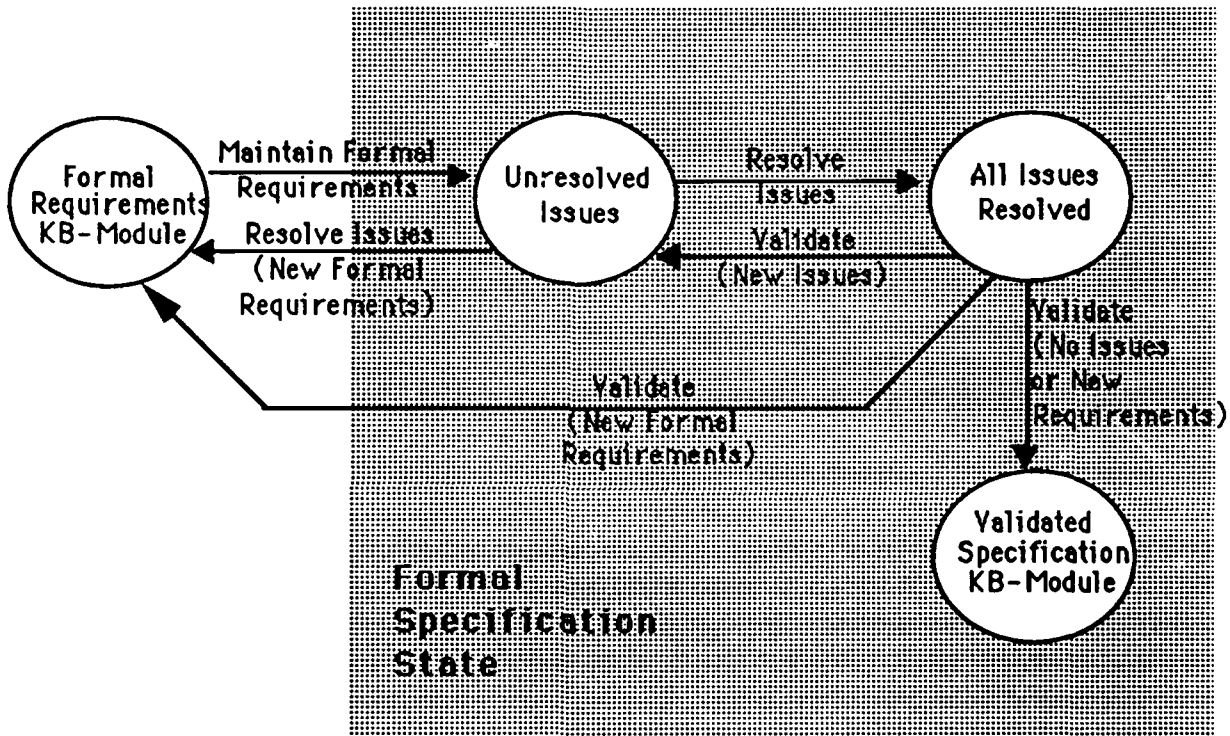


Figure 5-8: Sub-States of a Formal Specification KB-Module

- If all validation steps are completed and no new issues or requirements have been created, the kb-module is in the validated specification kb-module state. Alternatively, the results of validation can move the kb-module back into previous states. For example, new issues may be created during an unsuccessful validation, returning the kb-module to the unresolved state.

As shown in Figure 5-9, validation includes syntactic analysis, resource analysis, paraphrasing, and simulation. Syntactic analysis identifies low-level errors such as inconsistent use of types. Resource analysis checks the access patterns of objects in the kb-module against defined protocols (e.g., exported objects and used kb-modules). Paraphrasing entails the generation and user review of an English-like paraphrase of the specification. Simulation includes the definition,

execution, and review of simulated system behavior, derived from test cases for the specification.

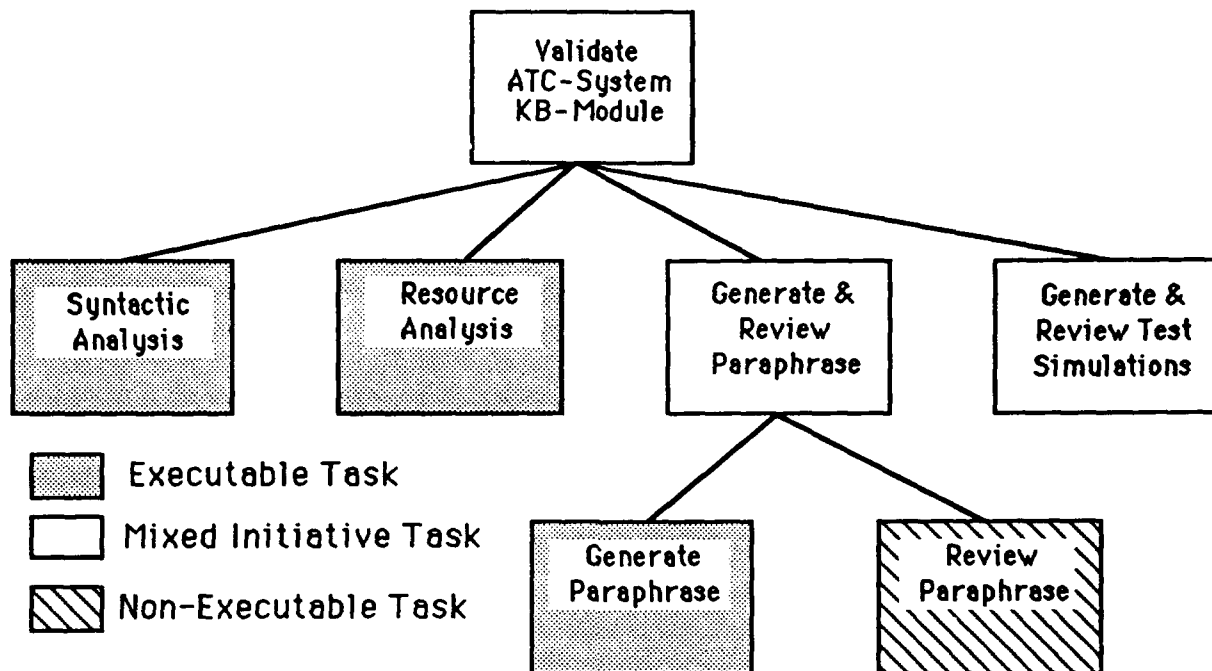


Figure 5-9: Sub-Tasks of Validate Specification

Returning to our example, suppose that the atc-system kb-module is currently in the "Resolved" formal specification state. Since the developer's goal is to move the kb-module to the validated state, the Concept Demo instantiates a task hierarchy to validate the kb-module. Figure 5-9 shows this hierarchy, and highlights its mixed initiative nature. The Validate ATC System task includes both executable and non-executable sub-tasks. Syntactic Analysis and Resource Analysis are both executable. Upon a command from the user, the Concept Demo can determine if any syntactic or resource conflicts are present, and will create issues to describe those conflicts. But while the system can generate an English-like paraphrase of the specification, it cannot review that paraphrase and determine how well it matches the intentions of the developers and users.

We use heuristics to define the order in which sub-tasks are performed. Resolving problems at the syntactic level may lead to the discovery of new problems at the resource level, so syntactic analysis precedes resource analysis. Similarly, we perform all executable validation tasks before mixed initiative tasks. This lets the Concept Demo identify and help resolve as many issues as possible before the specification is reviewed by human domain experts and executives, whose time is in high demand.

In our example, suppose that the atc-system kb-module successfully completes Syntactic Analysis. This analysis and its successful outcome are recorded in the

Concept Demo's History Mechanism, which automatically maintains a record of the sequence of development operations and the knowledge-base state on which they were performed (DeBellis, 1990). In Resource Analysis, however, an illegal access by atc-system of the atc-domain kb-module is discovered. As currently defined, methods and demons in the atc-system specification access the aircraft-location slot on instances of the aircraft class. Since the aircraft-location slot has not been exported from the atc-domain kb-module, the interface conventions of the atc-domain kb-module have been violated (see Figure 5-10). This violation is detected by the Resource Analysis, and an issue is created to mark this problem. The execution of the analysis, its unsuccessful outcome, and the specific issue created are captured by the Concept Demo History Mechanism.

The creation of a new unresolved issue has moved the atc-system kb-module back into the unresolved state. From the description of the issue and its knowledge of task postconditions, the Concept Demo can propose two alternative resolutions. The first resolution is to apply the export-object transformation. This would export the aircraft-location slot from the atc-domain kb-module. The second resolution is to apply the splice-communicator transformation. This would require the user to specify a communication slot or method which was visible to both kb-modules and could be used to communicate the value of the aircraft-location slot to the atc-system kb-module.

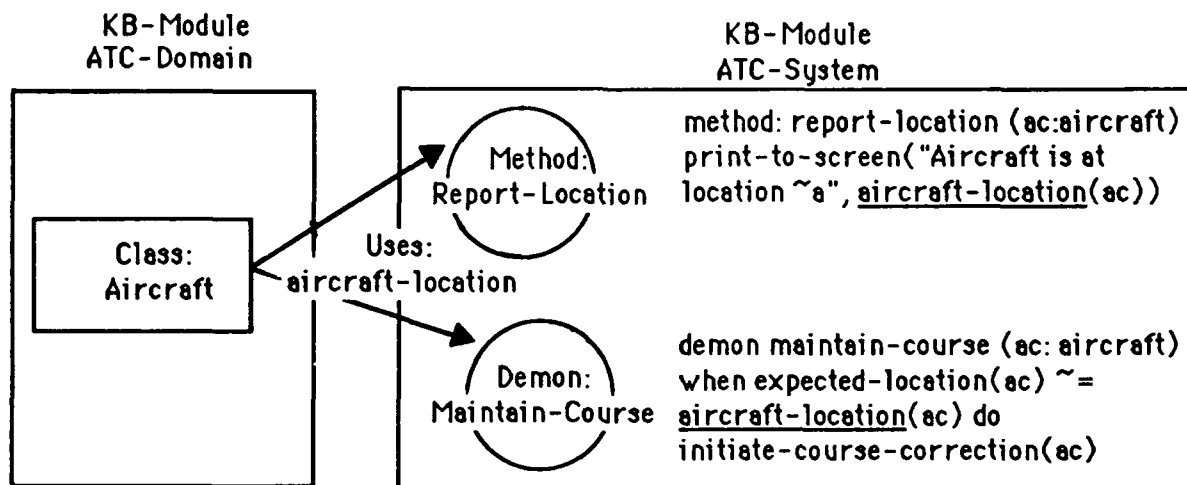


Figure 5-10: Illegal Access of Aircraft-Location Slot

An aircraft's location is only known to the real-world ATC System through intermediary processes such as radar tracking or visual observation. The developer therefore chooses to apply the splice-communicator transformation, as its effect will more closely correspond to the domain. The Concept Demo then adds a new class (radar) to the ATC-Domain kb-module. It also adds two new slots:

- radar-unit, which points from an aircraft to the radar on which it is monitored, and
- radar-track, which holds a representation of the location of the aircraft.

Another result of the splice-communicator transformation is that the radar-track and radar-unit slots are automatically exported from the atc-domain kb-module. In addition, all the methods and demons in the atc-system kb-module which accessed the aircraft-location of aircraft have now been modified to access the radar-track of the radar-unit which monitors that aircraft. Figure 5-11 illustrates the resulting situation. Again, the enactment of the Splice-Communicator transformation and its successful result are captured automatically by the Concept Demo History Mechanism.

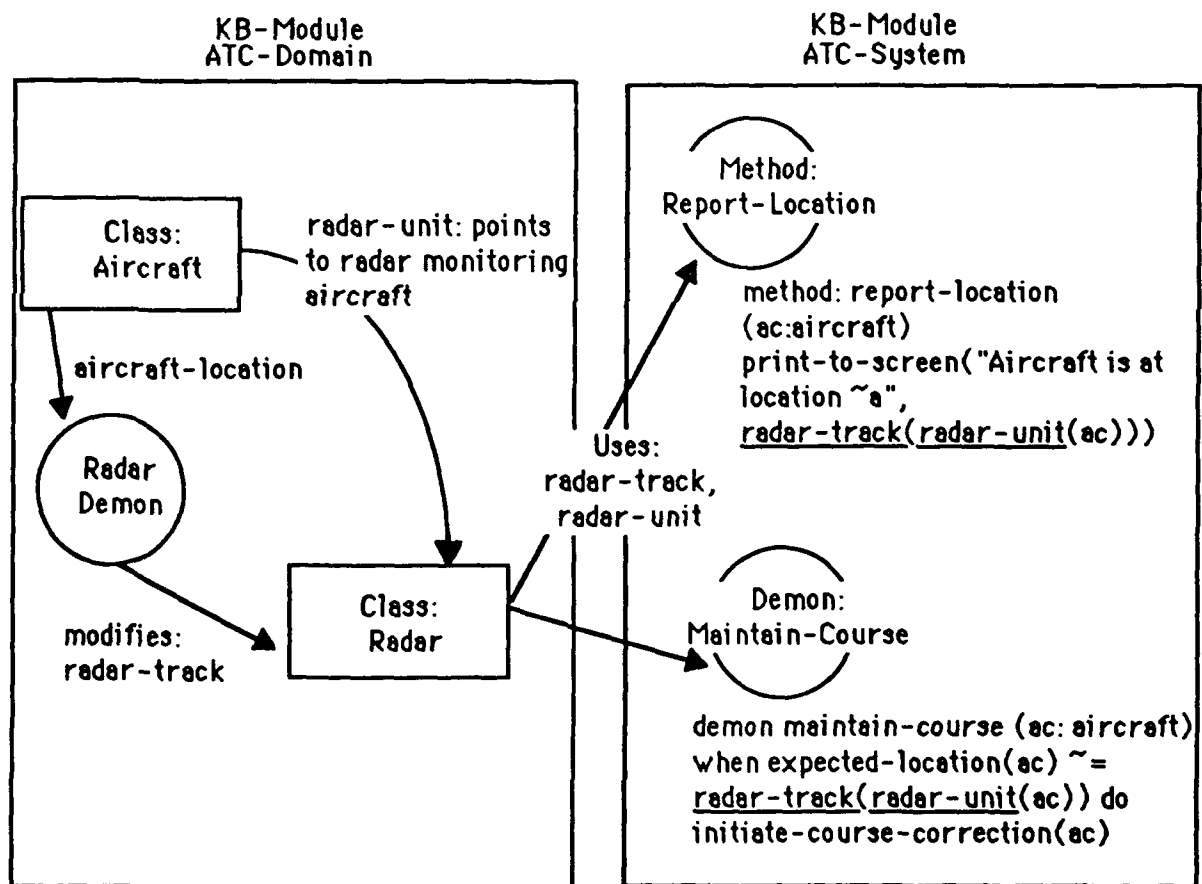


Figure 5-11: Resource Conflict Resolution

5.2.4 Discussion

The example above shows how the synthesis of AI planning techniques and knowledge-based software development can provide the benefits we discussed in this section's introduction.

- **Stronger guidance in transformation selection:** Because our process model formalism defines the set of possible kb-module states, the Concept Demo can determine the current-goal state for any kb-module. Further, since the formalism includes explicit definitions of specific transformations, the Concept Demo can identify candidate solutions with preconditions that match the kb-module's current state and post-conditions that match its desired state. Thus, in the example above, the Concept Demo Agenda suggests two candidate transformations which can resolve the illegal access issue.
- **Better development process traceability:** At the completion of each development step in our example, the Concept Demo History Mechanism automatically recorded the action taken, its outcome, and the resulting state of the knowledge-base. In the case of the Splice-Communicator transformation, this included information describing its rationale (i.e., performed to resolve illegal-slot-access and a reference to the hypertext description of the illegal-slot-access), providing strong traceability.
- **Higher-level development operations:** As illustrated in the example, the Splice-Communicator transformation creates a new class, defines two key slots on that class, and modifies existing methods and demons such that they access the appropriate slot on the new class. This is all performed in a single operation, with strong consistency maintenance capabilities, illustrating the power of transformation-based software development.
- **Finer-grained planning capability:** In most conventional environments, the AI-planner's recommended response to the illegal-slot access issue would have been "edit ATC System data-flow model" to correct the issue. The Concept Demo, because its model includes strong descriptions of specific transformations, is able to suggest more specific responses (Export-Object or Splice-Communicator).
- **Mixed-initiative development:** As Figure 5-9 shows, our formalism explicitly models executable, non-executable, and mixed-initiative tasks. In the example itself, the developer is presented with suggestions as to possible resolutions for the illegal-slot-access issue, but retains the ability to select the appropriate one. Further, at the developer's discretion, he can use a completely different technique that may be more appropriate in a specific development situation.

Every modeling formalism has its advantages, so it is important to examine each one against more general criteria, as well as against its own claims. We will now

evaluate our formalism against a set of process modeling criteria proposed by Chroust and reported by MacLean (1988). Chroust identified four important characteristics to be considered when evaluating a process modeling formalism.

1. Humans should be able to understand the model(s), both in simplified and in complete versions. The basic concepts in our model — task and state — are intuitively meaningful and familiar to most software development professionals. Their formal representations, as shown in the Appendix, can be interpreted meaningfully after a brief orientation to ERSLa syntax. As envisioned in the original KBSA statement (Green et al, 1983), the Concept Demo employs mixed-initiative interaction, enabling the developer and the system to complement each others' knowledge and expertise.

Further, as suggested by Kellner and Hansen (1988), we use generally accepted representations (both graphical and text-based) and present multiple, complementary views of the process to enhance the model's intelligibility. For example, we use trees to show structural relationships in the product and process hierarchy models. We use state-transition diagrams to depict the possible sequence of states a kb-module will evolve through as it is elaborated, although this kb-module state information is available only in text display in the Concept Demo delivery version. The text-based agenda is also familiar to most users. In all these presentations, the user has the ability to select the level of detail that is most relevant to the task at hand.

Perhaps most importantly, the transformations modeled in our formalism are defined as meaningful software engineering operations (e.g., splice-communicator) rather than document-based editing (e.g., insert-character). And the plan-based aspect of the model enables the association of transformations with both local and higher-level objectives, to provide even stronger information about the model's underlying logic. This helps our model satisfy Chroust's human understanding issue.

2. The model(s) should be interpretable by machines (e.g., computers). Since all the elements of the Concept Demo process model are expressed in a formal, executable language, ERSLa, our formalism satisfies Chroust's second issue, machine intelligibility. Indeed, the integration of the formalism with the powerful automated support provided by transformation-based development satisfies the requirement that the modeling approach "offer automated tools supporting the approach" (Kellner and Hansen, 1988, p. 27).

3. The model(s) should be adequate for use in industrial software development context, and should include the ability to express information about the necessary ancillary services. The adequacy of our model for use in an actual industrial environment has yet to be demonstrated. Its only application to date has been in a single-user, concept demonstration prototype system. The large amount of knowledge stored about each development object and task may

require the use of hardware and system software environments considered non-standard in industry. As discussed above, we also need to model multi-user development capabilities in the model prior to its application in a large-scale, real-world development project.

4. The model should be easy to manipulate, including the ability to modify, compare, and combine process models. Our formalism enables the manager to create, tailor, and combine process models in a flexible fashion. First, the plan structure has been developed in an object-oriented environment, bringing that approach's powerful support for reuse and adaptation. Second, the transformation-based development paradigm allows the capture and replay of transformation sequences (cf. Goldberg et al, 1989). Most importantly, the AI planning paradigm itself, as noted by Huff (1988), supports these characteristics as shown by capabilities such as failure recovery, detection and resolution of conflicting actions, identification of shortcuts, and goal revisions.

Our model satisfies Chroust's first, second, and fourth points, but the jury is still out on point three (suitability for industrial application). We regard the current Concept Demo process model formalism not as a destination, but rather as a promising avenue for further exploration of process model representations and the demonstration of the feasibility of the KBSA vision.

5.2.5 Possible Extensions of the Current Formalism

There are three ways in which the Concept Demo delivery version does not fully implement the model described above. First, our work has focused only on sophisticated guidance for an individual developer. However, as discussed above, we believe that we can provide multi-user capabilities similar to those developed by Mi and Scacchi (1990) by integrating their process representation meta-model with ours. This would also enable us to add a process model query facility like that in their Articulator. In addition to the current actual status information available via the Concept Demo's general object display mechanism, this would enable the user to get information about previous model states (development history) and possible model states (a what-if capability).

Second, to provide guidance to the developer, the current system relies exclusively on stored knowledge describing the interdependencies of development tasks and possible states of products. We did not integrate a theorem prover, but consider feasible and desirable the possibility of adding a direct inferencing facility such as that provided in the KBSA Development Assistant (Smith, 1991).

Finally, the Concept Demo delivery version offers only limited capabilities supporting the reuse of process models, although strong potential for this exists. The reuse and replay approach suggested by Goldberg et al (1989) and the replanning capabilities discussed by Huff (1988), can be used to develop

transformation-like capabilities for the adaptation of individual process models, enabling their continued use in dynamic environments as well as their revision for employment in different contexts.

Appendix 5.2-A: Splice-Communicator Task Definition and KB-Module State Descriptions Expressed in Extended Refine Specification Language

Task Definition: Splice Communicator

```
task splice-communicator
  subclass-of evolution-transformation
  subtask-of RESOLVE-ILLEGAL-SLOT-REFERENCE
  input-products (referenced-module: kb-module,
    referencing-module: kb-module, illegally-accessed-slot: slot)      preconditions
    {assigned-developer (kbm) = current-user ,
    checked-out-to (kbm) = current-user,
    illegally-accessed-slot in owned-objects(referenced-module) &
    ex(x) x in owned-objects(referencing-module) &
    illegally-references(x, illegally-accessed-slot) },
  postconditions
    {forall(x) x in owned-objects(referencing-module) =>
    legally-references(x, illegally-accessed-slot)}
```

KB-Module State Definitions

```
function empty-module?: map(kbm: kb-module) :boolean =
  empty(owned-objects(kbm))

function preformal-module? (kbm: kb-module) :boolean =
  ex(so) (so in owned-objects(kbm) & preformal-requirement?(so))

function formal-requirements-module? (kbm: kb-module) :boolean =
  ~preformal-module?(kbm) &
  ex(so) (so in owned-objects(kbm) & unmaintained-requirement?(so))

function formal-specification-module? (kbm: kb-module) :boolean =
  ~empty-module?(kbm) &
  ~preformal-module?(kbm) & ~formal-requirements-module?(kbm)

function optimized-module? (kbm: kb-module) :boolean =
  formal-specification-module?(kbm) &
  forall(x)(x in owned-objects(kbm) => checked-for-optimizations?(x))

function formal-requirement?(obj: object) :boolean =
  invariant?(obj) or function-with-conditions?(obj)

function preformal-requirement?(obj: object) :boolean =
  hypertext-requirement(obj) & empty(formalized-by(obj))

function unmaintained-requirement? (obj: object) :boolean =
  formal-requirement?(obj) & empty(maintained-by(obj))
```

6. Recommendations and Conclusions

This section describes our recommendations and conclusions based on the overall Concept Demo effort.

6.1 Recommendations for Future Design Decisions and Standards

In this section we address the requirements in sections 4.2.6 of the Concept Demo SOW. Section 4.2.6 requires documentation of:

- "... design decisions that should be considered by future efforts initiated under RADC's KBSA program, and shall address, as a minimum:*
- a. Enhancement of hardware platform independence.*
 - b. The need for consistent interface definitions and communication protocols.*
 - c. Environment services that should be present at different levels of granularity to support both commonality in usage and specialization.*
 - d. KBSA-environment extendibility that supports the integration of new functionality in a uniform way."*

In section 6.1.1 we discuss the use of commercial software. This was one of the most significant design decisions in the Concept Demo and is relevant to hardware independence of future KBSA systems (point a. above). In section 6.1.2 we describe the support environment for the Concept Demo and how that environment could be generalized to a KBSA support environment (point c.). In section 6.1.3 we discuss emerging interface and communication standards for Software Engineering Environments and how those standards could be used to build a KBSA-environment in an extendible way (points b. and d.).

6.1.1 Use of Commercial Software

One of the most significant design decisions made in the Concept Demo project was the decision to build the system on a commercial off the shelf (COTS) platform (Refine). The following describes pros and cons of using COTS tools in light of our experience with the Concept Demo. We first discuss the two arguments most frequently made against the use of COTS tools and then discuss lessons learned from the Concept Demo that are relevant to these arguments.

The standard arguments heard against the use of COTS tools involve cost and flexibility.

1) Cost. COTS technology increases the cost of both the development and run-time environments for a research prototype system. This can take away resources from the research effort and make it more difficult to distribute the prototype.

2) Lack of flexibility. Researchers do not like to be constrained by using tools that do not provide source code. COTS tools may enforce constraints on the implementation of certain types of functionality.

For the following reasons we believe that these arguments are often given too much weight when considering whether or not to use a COTS platform.

1) Cost of COTS versus cost to rebuild functionality. When considering the cost of using a COTS development environment, the cost for developers to build the functionality that already exists in the COTS tool — as well as the increased productivity that will be gained from using a more robust and well documented COTS development environment — must be factored in. When this is done, it will almost always be the case that the cost of the COTS tool is significantly less expensive than the cost to rebuild even a subset of its functionality.

2) Cooperation of COTS vendors. Most of the vendors of technology related to KBSA are eager to participate in research projects and to see prototypes developed using their environment. This participation increases their visibility and gives them access to new potential customers interested in advanced technology. Cooperative participation of vendors can result in discounts for run-time versions of prototype systems. In addition, vendors are often willing to provide normally hidden information to researchers allowing them to extend or modify the functionality of the COTS tool itself. For example, Reasoning Systems provided significant discounts for limited-use Refine licenses for technology transfer of the Concept Demo. In addition, Reasoning on several occasions provided extremely valuable support to help Concept Demo developers understand and modify internal features of the Refine language and environment.

3) Hardware and operating system portability. Building KBSA functionality on a COTS platform should significantly increase the portability of future KBSA systems. COTS vendors have a significant incentive to port their tools to various platforms and to minimize the work that must be done to port applications built using their tools from one supported platform to another. For example, commercial interface building packages such as Open Interface™ from Neuron Data allow developers to create interfaces that can be ported across various window managers and operating systems. These interface builders typically allow developers to create interfaces that can easily port to and from Motif, Open Look, Microsoft Windows, the Macintosh Interface, etc. Use of such interface-building tools and other COTS tools such as object-oriented databases and expert system shells would significantly increase the platform independence of future KBSA systems.

6.1.2 KBSA Environment Services

Figure 6-1 illustrates the general functionality developed for the "CD Framework". As described in section 2.3.1, the CD Framework consists of functionality that spans the software development life-cycle. In the Concept Demo it was used to integrate existing KBSA systems and as a foundation for requirements, specification and process guidance functionality.

ERSLa	Process	Presentation-based Interface
Dialect	Representation	Intervista
Refine KB		Common Windows
Common Lisp		
Unix & X Windows		

Figure 6-1: Block Diagram of the Concept Demo Framework

Req/Spec Languages	Knowledge-Based Process Management	Presentation-based Interface
Language Meta-Models		
Language Modeling	Extended Transactions & Configuration Management	Window Utilities
KB Language		
Programming Languages & OODB		
Operating System & Window Manager		

Figure 6-2 Block Diagram Proposing General KBSA Service Layers

Figure 6-2 shows a block diagram that is a generalization of the services provided by the Concept Demo framework. Figure 6-2 also adds services that were not required in the Concept Demo because it was primarily a single-user system. In the following we describe the layers in figure 6-2 and, where appropriate, describe commercial technologies that provide some or all of the functionality required for a particular layer.

Operating System & Window Manager. Recent KBSA efforts seem to have achieved a de-facto standard for hardware, operating systems and windowing environments. Recent KBSA projects by Software Options, Kestrel, ISI, Sanders, and Andersen Consulting have used Unix (almost exclusively Sun) workstations running X windows. We believe that this is a good standard to adhere to for the near future.

Programming Languages & OODB. The Concept Demo was implemented exclusively on top of Common Lisp. Future KBSA systems will need to support multiple programming languages – at the very least to test code generated for

run-time systems and most likely for the various tools to support the development environment as well. In addition, an object-oriented database will be essential to provide knowledge-base support for multiple users. This level should utilize commercial environments to support languages such as C++ and Lisp and a commercial object-oriented database.

KB Language. This corresponds to the "Refine KB" layer in figure 6-1. This layer consists of capabilities found in object-oriented and rule-based systems – classes, inheritance, message passing, inference engines supporting forward and backward chaining, etc. Examples of this layer in other KBSA environments are AP5, Socle, and Loglisp. There are several commercial tools on the market that provide sophisticated KB language support implemented in Lisp and C. These include Refine, Prokappa, and Art-IM.

Language Modeling. This corresponds to the "Dialect" layer in figure 6-1. This provides the ability to specify grammars which correspond to object meta-models for the language. That is, when parsing a language construct, the parser generated by a KBSA language modeling tool will create a parse tree that is a composite of instances of objects in the meta-model. For example, in Refine, parsing a function will create an instance of the class "function". That instance will have slots that point to the parameter list, the return type, and the body of the function. The value of each of these slots will in turn be an instance of another object in the meta-model which will further be decomposed into slots and values corresponding to the parse tree of the function. This object-based representation of parse trees makes writing general purpose analyzers and transformations much easier. We also include the transformation engine in this layer since it should be tightly coupled with the parser generator (e.g., the parser generator should output patterns that can be used by the transformation engine). The other example of this layer in current KBSA environments is the POPART system from ISI.

This language modeling capability is one of the most important new technologies to emerge from the KBSA effort to date. This is the primary reason that KBSA has had a big technology transfer impact on re-engineering – language modeling can be used to model and analyze systems in existing programming languages. The only commercial tool that currently provides these capabilities is the Dialect component in Refine.

Extended Transactions and Configuration Management. This layer is not present in the Concept Demo because the Concept Demo is primarily a single-user system. This layer extends the object-oriented database with capabilities required for group design environments. Gail Kaiser describes the capabilities needed for this layer in [Kaiser 88b]. Examples of KBSA environments that provide such capabilities are the Artifacts system [Karr 89] and the configuration management system in the KBSA Framework [Larson 90]. Some of the better object-oriented database systems provide a subset of this layer. We believe that

the best approach for KBSA is to build this layer on top of a commercial object-oriented database.

Window Utilities. This layer corresponds to the Common Windows/Intervista layers in the Concept Demo and to "Level 1" (Building Blocks) in the KBSA User Interface Environment (KUIE) [Larson 90]. This layer provides an object-oriented front end to window manager functionality and also provides general classes useful for software development environments such as icons, links, etc. There are several C and C++ based commercial tools that provide most of the functionality required for this layer. As described above, one of the advantages of using such tools is that they provide support for porting user-interfaces to various operating systems and window managers.

Language Meta-Models. This layer consists of meta-models for programming and requirements and specification languages. In the Concept Demo, this layer is implicit in the ERSLa layer. We make it an explicit layer here to emphasize that future KBSA environments will contain meta-models of programming languages used for code generation as well as various requirements and specification languages. There are currently no commercial implementations of this layer (or any of the following layers).

Requirements and Specification Languages. This layer corresponds to the ERSLa layer in the Concept Demo. We use the plural here to emphasize that future KBSA environments will eventually contain multiple specification and requirements languages. In fact, even in the Concept Demo there were small grammars in addition to ERSLa for describing hypertext networks and kb-modules. There are currently no commercial implementations of this layer.

Knowledge-Based Process Management. This corresponds to the task classes and methods inherited from the PMA and extended in the Concept Demo. Other examples of process representation formalisms are transaction graphs [Karr 90] and the rules in Marvel [Kaiser 88]. There are currently no commercial implementations of this layer.

Presentation-based Interface. This corresponds to the presentation-based interface described in detail in section 2.3.1.1. and to level 2 in KUIE. Other examples of presentation-based interfaces are the interfaces in the KBRA [Harris 88] and Aries [Johnson 90]. There are currently no commercial implementations of this layer.

6.1.3 Interface Definitions and Communication Protocols

The KBSA community should not invent its own interface definitions or communication protocols. There is significant progress being made in defining Open Systems and CASE standards. KBSA should pay close attention to these standards and adhere to them as much as possible. Adherence to such standards

will allow KBSA systems to smoothly incorporate existing conventional tools such as editors, compilers, etc. Adherence to such standards will also allow existing conventional environments to incorporate individual KBSA modules with functionality such as language modeling, transformations, specification languages and analysis, etc.

The CASE standard that currently seems the most promising is PCTE. It has several factors in its favor:

- 1) It is object-oriented and thus will not hinder the development of a knowledge-based repository the way a relational repository standard would.
- 2) It is an emerging standard for software engineering tools used by the DoD. For example, Cadre, Digital, Hewlett-Packard, Sun, and Bull all have CASE tools that are committed to using PCTE. In addition the STARS program and the new version of SLCSE being developed by ISSI for Rome Laboratory plan to be PCTE compliant.
- 3) It has recently been adopted by IBM as part of IBM's AD-Cycle CASE standard.

6.2 Lessons Learned: Technology Transfer

As of September 30, 1992, nine organizations had acquired KBSA Concept Demo licenses and ten other licenses were in process. To gauge the effectiveness of the Concept Demo with respect to its technology transfer objectives, we conducted a survey of the licensee organizations to learn about their experiences with the Concept Demo. Similarly, we conducted a survey to understand what issues had slowed the completion of the licensing process in the organizations that had not completed the licensing process. A full description of the responses is provided in Appendix A of this report.

Of the nine licensees, we received responses from seven (78% response rate). These organizations are identified using the following coding scheme:

Code	Organization Description	CD Training?
L1	DoD Graduate Institute (U.S.)	Yes
L2	Non-profit Analysis AI Research Center (U.S.)	No
L3	Industrial (Petroleum Services) CS Center (U.S.)	No
L4	Industrial (Systems) Development Group (U.S.)	No
L5	Industrial (Telecom) R&D Center (U.S.)	Yes
L6	University Computer Science Dept (Middle East)	No
L7	Federal (Non-DoD) Technology Institute	No

Two licensees did not respond: a non-profit research center (U.S.) and an industrial (defense) software technology group (U.S.). Of the ten organizations currently in the licensing process, we received feedback from eight of them (80% response rate). These organizations are coded as follows:

Code	Organization Description	Refine?
N1	University Research Center (U.S.)	No
N2	University Research Center (U.S.)	No
N3	University Research Center (FDR)	No
N4	DoD Training Center	No
N5	DoD Research Agency	No
N6	University Research Center (U.K.)	No
N7	Industrial (Telecom) Technology Center (US)	No
N8	Non-Profit Research Institute (U.S.)	No

Two in-licensing-process organizations did not respond to the survey: a university computer science group (FDR) and a non-profit software engineering center (U.S.).

A representative of N5 attended the Concept Demo training program, but is no longer involved in that organization's Concept Demo acquisition efforts. None of the other in-license-process organizations have been represented at Concept Demo training programs.

The feedback we received appears to support our claim [Sasso 1991] that the transitionability of KBSA technology can be projected using the technology transfer model proposed by [Rogers 1983]. Rogers suggests that the perceived presence of four factors (relative advantage, compatibility, trialability, and observability) will encourage the transfer of a new technology, while its perceived complexity will discourage adoption.

Relative Advantage: The greater the extent of the new technology's perceived superiority over existing technologies, the greater will be its likelihood of adoption.

Demonstrations have been reported by two of the licensees (L1¹ and L4). The demonstrations conducted by L1 appear to have generated enthusiastic support for and interest in KBSA technology. Those conducted by L4 appear to have been less successful in this regard. As more licensees conduct demos and report the reactions of their audiences, we hope to identify and report the factors that are associated with more successful demonstrations.

¹The codes used in this section index survey respondents (ie.g., L1) and their specific comments (e.g., L1-4. denotes the response by organization L1 to question 4). These comments are presented in Appendix A of this report.

Compatibility: The greater the consistency between the values, approaches, and technical infrastructures required by the new technology and those of its potential adopters, the greater its likelihood of adoption.

Trialability: The greater the extent that the new technology can be tried out in an experimental, low-risk situation, the greater its likelihood of adoption.

From the information reported above, compatibility and trialability appear closely related in this case. Where the technical infrastructure of the Concept Demo (specifically, the Software Refinery environment) is compatible with that of the licensee, the stakes involved in trying the Concept Demo out are perceived as much lower (see comments N1-2, N2-1, N2-3, N3-1, N4-1, N5-1, N6-2, and N7-2). There appear to be two components of this increased hassle of licensing: bureaucratic organizational license approvals and processing (see N3-1 and N5-1) and the additional cost of acquiring the Software Refinery and Franz Allegro Common Lisp (see N1-2, N2-1, N2-3, N4-1, N6-2, and N7-2). Similarly, one licensee reported that previous experience with Refinery facilitated the installation process (L4-1).

These comments underline the wisdom of Rome Laboratory in its inclusion in the Concept Demo statement of work the requirement that transferable licenses for software necessary to run the Concept Demo be delivered as part of the project deliverables. These should facilitate the diffusion of KBSA technology, by enabling Rome to reduce the up-front investment required to take the Concept Demo for a "test drive."

Observability: The greater the degree to which the benefits of the technology's use are visible to the adopter, the greater the likelihood of its adoption.

Complexity: The greater the extent to which the users perceive the new technology as difficult to learn and use, the *lower* its probability of adoption.

These preliminary comments also suggest that observability and complexity can be discussed together in this analysis. Only four of the responding licensees (L1, L2, L4, and L5) have reported actual hands-on experience, and one of these (L5) is very reticent in his responses. Comments made by L1 (L1-3 and L1-7) and L2 (L2-7) suggest that the impact of KBSA technology is observed fairly easily in the Concept Demo, while L4's audience (L4-3) appears less convinced. L1, L2, and L4 all suggest that the Concept Demo documentation could be improved (L1-7, L2-7, L4-2, L4-2).

Only (L1) can describe working with the technology in a more free-form fashion. Its comments (L1-4 and L1-5) are positive, but tentative.

General Discussion: L1 has clearly reported best experience and demonstration results, and is clearly the most experienced and familiar with the Concept Demo. Representatives of this organization attended a Concept Demo Training Session. This suggests that the Concept Demo can clearly serve as an effective vehicle for communication of the KBSA paradigm, but that significant familiarity (with both the paradigm and the CD itself) is a prerequisite for effective demonstration and explanation.

6.3 Conclusions

In this section, we summarize the major accomplishments and lessons learned from the Concept Demo.

6.3.1 KBSA Integration

The Concept Demo is the first KBSA system to demonstrate coverage across the life-cycle — from informal requirements to formal specifications to code. It is also the most significant integration to date of previous KBSA concepts and technology.

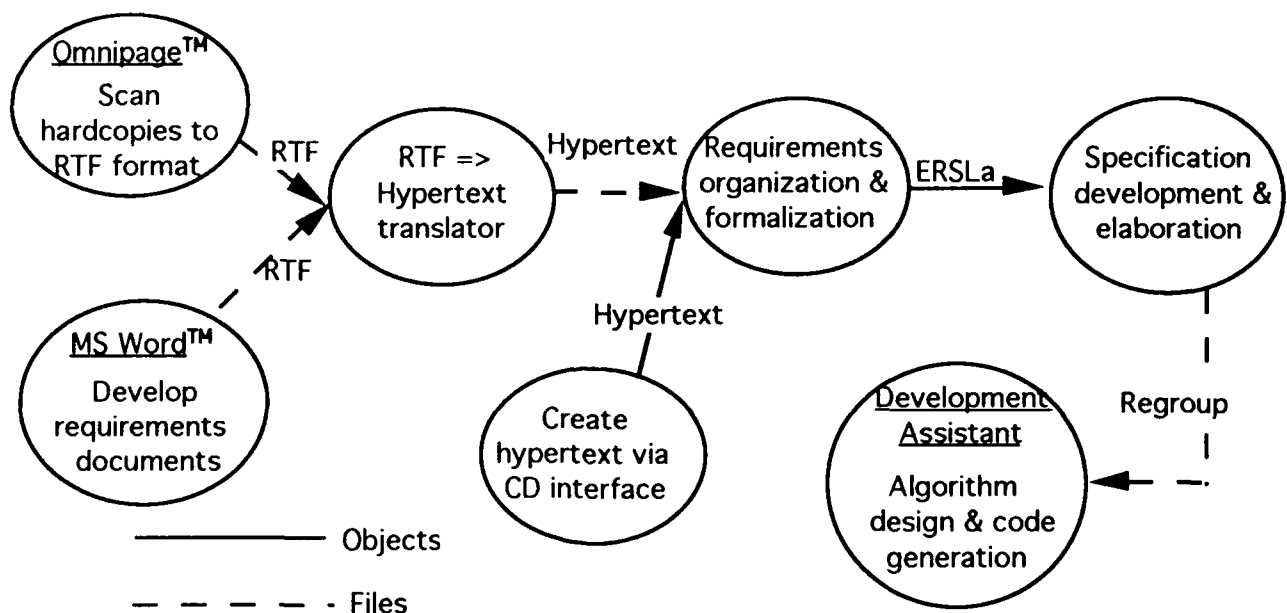


Figure 6-1: Life-Cycle Coverage in the Concept Demo

Figure 6-1 illustrates the life-cycle coverage of the Concept Demo. Informal hypertext requirements can be developed either by scanning hardcopies of existing requirements documents, using a WYSIWYG editor such as Microsoft Word, or using the Emacs interface provided by the Concept Demo. The informal requirements can then be formalized and elaborated using evolution transformations and specification feedback. Finally, the resulting system can be input to the Development Assistant which can transform the specification into efficient code.

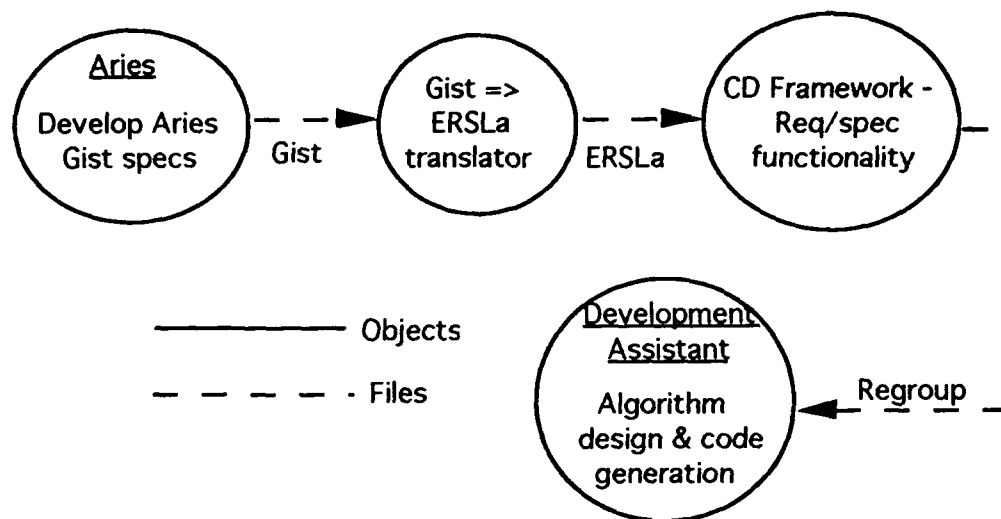


Figure 6-2: KBSA Integration in the Concept Demo

Figure 6-2 highlights the integration of KBSA technology in the Concept Demo. The Concept Demo provided the first integration of a system built at ISI (the Aries system) and Kestrel (the Development Assistant). This integration was made possible by the cooperation of ISI, Kestrel, and Andersen Consulting.

- ISI developed a new version of Gist as part of the Aries project. The Aries version of Gist used a syntax similar to the Refine language. ISI also developed a translator that translates most constructs from Aries Gist into ERSLa.
- Kestrel developed the Regroup language as a superset of the Refine language and made it available to Andersen Consulting a year before the completion of the Development Assistant project. The previous specification language used in the Performance Estimation Assistant (Performo) did not support Refine capabilities such as the assignment of slot values to knowledge-base objects. These capabilities are essential to languages such as Gist and ERSLa.
- Andersen Consulting developed the ERSLa Language which extended the Regroup language with constructs from Gist such as invariants and demons.

The technical integration of Aries and the Development Assistant is only the first step to true KBSA integration. This technical integration pointed out that the style of specification supported by Aries is significantly different from that supported by the Development Assistant. Aries specifications consist of a mixture of declarative constructs (e.g., invariants and pre/post conditions) and operational constructs (demons, functions, and procedures with bodies). Specifications for the Development Assistant must be purely declarative. In

addition, Development Assistant specifications must be written as domain theories. This requires the creation of a large set of distributive laws as a major part of the specification. It takes a significant amount of training (even by developers with strong backgrounds in logic and formal methods) to create such domain theories.

The advantage of the Aries approach to specifications is that it is more intuitive and easier for the average developer to use. The advantage of the Development Assistant approach is that it provides a rigorous distinction between the "what" and the "how" since the specification is purely declarative. Further, it supports the design of algorithms using meaning-preserving transformations. The reconciliation of these two approaches is a significant issue for future KBSA research.

6.3.2 Architecture for Intelligent Assistance

The Concept Demo was the first KBSA system to demonstrate intelligent assistance. Previous systems provided very sophisticated tools, but did not provide feedback or guidance to the developer on how to use those tools. The intelligent assistance for the Concept Demo included a general architecture that integrated concepts from process-based environments such as CLF and transformation-based environments such as the Specification Assistant. This architecture demonstrated the following:

- An integrated process-support environment capable of invoking traditional tools (e.g., editors) as well as transformations.
- An environment that supports traditional styles of work while providing the benefits of formal development techniques and process support. For example, developers can achieve the benefits of a formal specification language while editing and viewing the specification language through graphic interfaces.
- Integration of low-level tasks such as transformations with higher level project-management tasks. This automates many project management activities (e.g., updating the status of project-management tasks based on activities performed in the developers workspace).
- An open architecture that separates the user-interface, product/process analysis, product transformations, and process automation into independent modules.

The Concept Demo architecture is a design for a KB-CASE tool that could be built using existing technology. A KB-CASE tool would provide the ability to use transformations in combination with traditional tools (such as editors). Where appropriate, the power of transformations can be applied; where that technology

is not yet developed, conventional support can be provided. A KB-CASE tool would not require radical changes in the way developers work, but it would still provide them with many of the advantages of process-based and formal development environments. The open architecture of a KB-CASE tool would allow it to incorporate emerging mature technology from the KBSA program and to gradually evolve into a full-fledged KBSA.

6.3.3 A KBSA Process Model

Section 5 (above) has presented a high-level overview of one possible process model for software development in a KBSA environment. The model's major activities are organized around the degree of formalization and elaboration of the software development artifact. They need not proceed in lockstep, but rather provide the developer with strongly integrated facilities enabling rapid cycles of software development, validation, elaboration, and ongoing enhancement. The model recognizes that support is needed for the acquisition and organization of pre-formal requirements information as well as for the creation and extension of formally expressed specifications, and extends an earlier model to do so. It recognizes the importance of interfacing with existing systems and the recovery of procedural knowledge embedded within them, and provides these capabilities by inclusion of Software Reverse Engineering as a top-level activity.

By integrating an AI planning formalism with a knowledge-based software engineering environment, the Knowledge-Based Software Assistant Concept Demonstration System exemplifies significant advantages over conventional development support. These include (1) stronger guidance in the selection of development operations; (2) better development process traceability; (3) more powerful development support; (4) finer-grained planning capability; and (5) mixed-initiative development. This section will describe the planning formalisms used to accomplish this integration, and present an example illustrating these benefits.

6.3.4 Formal Validation Extension

The goals of the formal validation extension to the KBSA Concept Demo project were to study the techniques used in the KBSA paradigm to improve the software development process, identify development phases where formal validation is necessary, and then explore formal validation techniques that can be applied to those phases.

We found that there was a need for reducing the complexity of validating the specifications developed using the existing KBSA systems. We determined that the complexity of the validation task could be reduced if it could be broken down into incremental validation steps that accompany incremental evolutionary

development. Formal techniques can be applied to design evolutionary operations that support such incremental development and validation.

We accomplished the following in this project:

- Developed a conceptual model for incremental evolution and validation of specifications.
- Put the evolution transformations in a more general software reuse framework and discovered that retrieval and composition operations were useful for evolutionary development and were missing in the current KBSA.
- Designed formal operation for performing composition of specifications in an attempt to fill the gaps mentioned above.
- Implemented these formal operations to run on a subset of the formal specification language, ERSLa.
- Developed and implemented a scenario that demonstrates the use of the composition operations.

As part of future work we intend to design formal operations for component retrieval, and put the evolution transformations in the current Concept Demo system (the adaptation operations) in this formal framework. We will also study the interaction of retrieval, adaptation, and composition operations to guide developers in the specification development process.

7. Bibliography

Andersen Consulting, "KBSA Concept Demo User's Manual", Contract No. F30602-89-C-0160, CDRL Sequence No. A005, Beta Distribution version, July, 1992.

Balzer, R. A 15 Year Perspective on Automatic Programming. IEEE Transactions on Software Engineering. SE-11(11): November 1985.

Barghouti, N. and Kaiser, G. Multi-Agent Rule-Based Software Development Environments. Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference, pages 375-387. Syracuse, NY: 1990.

Barghouti, N. and Kaiser, G.E., "Concurrency Control in Advanced Database Applications", ACM Computing Surveys, Vol. 23, No. 3, September 1991.

Blaine L. , Goldberg et al., "Progress on the KBSA Performance Estimation Assistant", Proceedings of the 3rd Knowledge-Based Software Assistant Conference, 1988.

Boehm, Barry W. A Spiral Model of Software Development and Enhancement. IEEE Computer (May 1988).

Cabral, Gui and M. DeBellis, "Domain Specific Representations in the KBSA Concept Demo", Proceedings of the 6th Knowledge-Based Software Assistant Conference, 1991.

Cohen, D. *AP5 Manual*, USC Information Sciences Institute, 1985

Conklin, J., and M. Begeman. "gIBIS: A Hypertext Tool for Exploratory Policy Discussion." *Proceedings of the Conference on Computer Support for Cooperative Work*. Portland, OR: 1988, pp. 140-152.

Curtis, B. Empirical Studies of the Software Development Process (presentation). Software Process Symposium. Washington, DC: September 17-18, 1990.

Daum, M. and Jullig, R.J., "Knowledge-Based Project Management Assistant for Ada Systems", RADC-TR-90-418, 1990.

Day, W. Distributed Multi-Agent Planning (technical report). RADC-TR-90-410. December, 1990.

DeBellis, M., "The KBSA Concept Demonstration Prototype", Proceedings of the 5th Knowledge-Based Software Assistant Conference, 1990.

Feather, M. Detecting Interference When Merging Specification Evolutions. Proceedings, 5th International Workshop on Software Specification and Design. 1989.

Fikes, Richard, and Tom Kehler. "The Role of Frame-Based Representation in Reasoning", Communications of the ACM, 28(9): September 1985.

Gilham, Li-mei, Jullig, Richard, Ladkin, Peter, Polak, Wolfgang, "Knowledge-Based Software Project Management", Kestrel Institute Technical Report: KES.U.87.3, November, 1986.

Gilham, Li-mei, Goldberg, Allen, Wang, T.C., "Toward Reliable Reactive Systems", Proceedings of the 5th International Workshop on Software Specifications and Design, Pittsburgh, PA., May 1989.

Goldberg, A. L. Blaine, T. Pressburger, X. Qian, T. Roberts, and S. Westfold. KBSA Performance Estimation Assistant (final technical report). RADC-TR-89-98. August, 1989.

Goldberg, A.. Reusing Software Developments. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Goldman, N. Three Dimensions of Design Development. ISI/RS-83-2. July, 1983.

Green, C., Luckham, D., Balzer, R., Cheatham, T. and Rich, C., "Report on a Knowledge-Based Software Assistant," RADC TR 83-195, Contract No. F30602-81-C-0206, Kestrel Institute, Palo Alto, California, June 1983.

Guindon, R., H. Krasner, and B. Curtis. Breakdowns and Processes During the Early Activities of Software Design by Professionals. pages 65-82 in Empirical Studies of Programmers (Second Workshop). G. Olson, S. Sheppard, and E. Soloway, eds. New Haven, CT: 1987.

Harris, D. and Czuchry, A., "Knowledge Based Requirements Assistant", RADC-TR-88-205, 1988.

Huff, K. Plan-Based Intelligent Assistance: An Approach to Supporting the Software Development Process (doctoral dissertation). Department of Computer and Information Science, University of Massachusetts, September, 1989.

Huff, K. Representing Processes as Plans to Achieve Goals (presentation). Software Process Symposium. Washington, DC: September 17-18, 1990.

Information Sciences Institute, "Common Lisp Framework and Formalized System Development chapters in 1985 Annual Technical Report", December 1986, ISI Technical Report: ISI/SR-86-170.

Jagadish, H.V. and Qian, X., "Integrity Maintenance in an Object-Oriented Database", Proceedings of the 18th VLDB Conference, 1992.

Johnson, W. Lewis. Overview of the Knowledge Based Specification Assistant. Proceedings of the 2nd KBSA Conference. August, 1987.

Johnson, W. Lewis, et al., "The Knowledge-Based Specification Assistant, Final Report", Rome Air Development Center, Contract No. F30602-85-C-0221, 1988a.

Johnson, W. Lewis, and Yue, K., "An Integrated Specification Development Framework", Proceedings of the 3rd Knowledge-Based Software Assistant Conference", 1988b.

Johnson, W. Lewis, and M. Feather. Building Evolution Transformation Libraries. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Johnson, W., and M. Feather. Using Evolution Transformations to Construct Specifications. in Lowry, M. and R. McCartney, eds. Automating Software Design. AAAI Press: 1990.

Johnson, W. Lewis. Personal Communication on Partial Specification. August, 1990.

Johnson, Lewis, and Harris, Dave, "Requirements Analysis Using Aries: Themes and Examples", Proceedings of the 5th Annual Knowledge Based Software Assistant Conference. September, 1990.

Jullig, R. Progress on the Project Management Assistant. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Jullig, R., et al. KBSA Project Management Assistant. RADC-TR-87-78. July, 1987.

Jullig, R., M. Daum, and P. Ladkin. "Approaches to Planning in the Project Management Assistant." in Proceedings of the 3rd Annual Knowledge-Based Software Assistant Conference. Utica, NY: 1988.

Kaiser, G., P. Feiler, and S. Popovich. Intelligent Assistance for Software Development and Maintenance. IEEE Software (May 1988).

Kaiser, G.E., Barghouti, N.S., Feiler, P.H. and Schwanke, R.W., "Database Support for Knowledge-Based Engineering Environments", IEEE Expert, Summer 1988b.

Karr, M. and Holloway, G.H., "Beyond the Read-Eval Loop: Architecture of the E-L Environment", Unpublished paper from Software Options Inc., August 1989.

Karr, Michael, "Transaction Graphs: A Sketch Formalism for Activity Coordination", Rome Laboratory Technical Report, RADC-TR-90-347, December, 1990.

Kellner, M. and G. Hansen. Software Process Modeling. Technical Report CMU/SEI-88-TR-9 (May, 1988).

Kotik, G. and L. Markosian. KBSA for Automated Software Analysis, Test Generation, and Management. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Kozaczynski, W., and J. Ning. SRE: A Knowledge-Based Environment for Large-Scale Software Re-engineering Activities. Proceedings of the 11th International Conference on Software Engineering. May, 1989.

Larson, A., Kimball, J., Clark, J. and Schrag, B., "KBSA Framework Final Report", RADC Technical Report, RADC-TR-90-349, December 1990.

MacLean, R. Enaction Formalisms (session summary). Proceedings of the 4th International Software Process Workshop, pages 11-13. Moretonhampstead, Devon, UK: 1988.

Martin, J. and C McClure, Software Maintenance: The Problem and its Solution. Prentice-Hall. Englewood Cliffs, NJ: 1983.

Mi, P. and W. Scacchi. A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes. IEEE Transactions on Knowledge and Data Engineering 2(3): 283-294 (September 1990).

Mui, C., and M. DeBellis. KBSA Technology Transfer: An Industry Perspective. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

National Institute of Standards and Technology, "Reference Model for Frameworks of Software Engineering Environments", NIST Special Publication 500-201, December, 1991.

Newcomb, P. Knowledge-Based Reverse Engineering for Re-engineering and Reuse. Proceedings of the 4th Annual Knowledge Based Software Assistant Conference. September, 1989.

Osterweil, L. "Software Processes Are Software Too." Proceedings of the Ninth International Conference on Software Engineering, pages 2-13. Washington, DC: 1987.

Perry, D. Problems of Scale and Process Models. Proceedings of the 4th International Software Process Workshop, pages 126-129. Moretonhampstead, Devon, UK: 1988.

Ramesh, B. and Dhar, V., "Representation and Maintenance of Process Knowledge for Large Scale Systems Development", in Selfridge, P.G., ed., Proceedings of the Sixth Annual Knowledge-Based Software Engineering Conference. Syracuse, NY: 1991.

Reasoning Systems Inc., "Refine User's Guide", Palo Alto, California, June 15, 1986.

Rogers, E. The Diffusion of Innovation. Free Press: NY, 1983.

Sasso, W.C. Motivating Adoption of KBSA: Issues, Arguments, and Strategies. in Selfridge, P.G., ed., Proceedings of the Sixth Annual Knowledge-Based Software Engineering Conference. Syracuse, NY: 1991.

Sasso, W. and DeBellis, M., "A Software Development Process Model for the KBSA Concept Demonstration System", Proceedings of the Sixth Annual Knowledge-Based Software Engineering Conference. Syracuse, NY: 1990.

Smith, D., "KIDS: A Semiautomatic Program Development System", *IEEE Transactions of Software Engineering*, Vol 16, No. 9., September 1990.

Smith, D. "Development Assistant Final Report", Rome Laboratory Technical Report, KES.U.91.7, July 25, 1991.

Swaminathan, Kishore, "TAO: A Domain-Independent Approach to Information Extraction from Large Formatted Documents", Proceedings of the DARPA Document Understanding Workshop, Palo Alto, CA, May 1992.

Williams, Gerald B., and Jay J. Myers. Exploiting Language Metamodel Correspondences to Provide Paraphrasing Capabilities for the Concept Demonstration Project. Proceedings of the 5th Annual Knowledge Based Software Assistant Conference. September, 1990.

Williams, M. What makes RABBIT run? International Journal of Man-Machine Studies. 21:333-352. 1984.

Winograd, T. and F. Flores. Understanding Computers and Cognition. Addison-Wesley. Reading, MA: 1987.

Appendix A: Technology Transfer Survey Results

As of September 30, 1992, nine organizations had acquired KBSA Concept Demo licenses and ten other licenses were in process. To gauge the effectiveness of the Concept Demo with respect to its technology transfer objectives, we conducted a survey of these organizations to learn about the licensees' experiences. Similarly, we conducted a survey to understand what issues had slowed the completion of the licensing process in the organizations that had not completed the licensing process.

A.1. Feedback from Licensees

Of the nine licensees, we received responses from seven (78% response rate). These organizations are identified using the following coding scheme:

Code	Organization Description	CD Training?
L1	DoD Graduate Institute (U.S.)	Yes
L2	Non-profit Analysis AI Research Center (U.S.)	No
L3	Industrial (Petroleum Services) CS Center (U.S.)	No
L4	Industrial (Systems) Development Group (U.S.)	No
L5	Industrial (Telecom) R&D Center (U.S.)	Yes
L6	University Computer Science Dept (Middle East)	No
L7	Federal (Non-DoD) Technology Institute	No

Two licensees did not respond: a non-profit research center (U.S.) and an industrial (defense) software technology group (U.S.).

1. Were you able to access and install the Concept Demo successfully? (If not, please describe the nature of any problems.)

L1-1: I had no problem installing the system after using ftp to get it.

L2-1: Yes, we had one problem with certain windows coming up which was fixed via (re::reset-resource ri::*small-text-window-resource*) (on Andersen's advice).

L3-1: Bad News. I was planning on putting one of our Summer students to work on playing the demos, but then when the time came we found something more urgent for her to do. So ... we have not even installed the SW. I still hope to do it next time I have inexpensive labor around.

L4-1: The CD installed correctly the first time. It was relatively easy, but I'm familiar with the commands you used. We already had X11 and Refinery installed, so that helped.

L5-1: Yes.

L6-1: The disk I ordered has been installed yesterday, and I can finally try to install the KBSA CD, for which I got a license in May [5 months previous]. However, the pub/KBSA directory no longer exists ... Please send me new ftp instructions ...

L7-1: We have not yet accessed or installed the Concept Demo. The group that was interested in evaluating KBSA technology has been reassigned to perform another study, but expects to return to KBSA in another several months. At that point, we plan to install the technology. Please keep us informed of training programs and later releases.

2. Did you try to perform the pre-defined demonstration scenarios? If so, were you able to do so successfully? What types of problems (if any) did you encounter?

L1-2: We had no major problems with the scenarios. Occasionally, for reasons we could not figure out, the system would stall requiring getting out of the Concept Demo. This was very infrequent and could have been due to problems unrelated to CD.

L2-2: We eventually performed them all. At first we were slowed by the window resource problem. We also had problems running the simulation scenario which seemed to freeze after we had moved around a few of the windows (despite being warned against this :-)).

L4-2: We did run some of the scenarios. A few data input problems cropped up. One case was where we tried to create a new knowledge-base module before the directory existed, which was unsuccessful. Another was the use of decimal points in the Browsing scenario when trying to change the spacing for the depth of a tree. One thing I found perplexing was that the user manual specifies how to start a scenario, and how to end it, considering the comment that Restore-a-scenario is to be used for a single demonstration. Moving from one scenario to another was not well explained, and shutting down the scenario with kill-refine seems to be overreacting.

L5-2: Yes, no problems.

3. Did you demonstrate the Concept Demo to others? If so, please give us a guesstimate of the number of people who have seen your demos and the nature of their work (e.g., SE practitioners, SE researchers, students, technical managers, high-level executives, ...). Can you say anything about the reactions of the audience to the demos?

L1-3: I and some of my students have demonstrated CD to over 100 people (mostly students and about 5 - 10 faculty). The audience has been very impressed with the system. The audience had very limited exposure to concepts of KBSE and the demos created a lot of interest about KBSE. Many commended the quality and stability of the implementation, given that CD is a demonstration system. I expect to have about 5 students working with the system, extending its functionalities or developing models using it.

L2-3: No, though not for lack of perceived interest. We had planned demos for two different groups, but schedules being what they are the demos were bumped.

L4-3: I demonstrated it to X, which was more of a team effort the first time. He seemed pleased with it and wanted to work with it more extensively. Most recently I demonstrated it to about 5 SE research managers, after talking about it to a group of about 10 SE research managers. The 5 out of 10 group appeared to be interested in it, but losing half the audience was annoying. The only interesting query was a question about what OOA/OOD methodology [is] supported by the CDS. What's your reply to that?

L5-3: Not yet, planned for Q1'93.

4. Have you tried to uses the Concept Demo functionality outside of the pre-defined scenarios? Please briefly describe any such experiences.

L1-4: We have tried to develop examples using CD, but this work has not been extensive.

L2-4: No.

L4-4: Not really.

L5-4: Not yet.

5. Have you attempted to build software extending the Concept Demo functionality? Please briefly describe any such experiences.

L1-5: We are working on incorporating a model of rationale (REMAP) extending the Semantic Network functionality of CD. The early attempts seem to suggest that we will be successful in our efforts.

L2-5: No.

L4-5: No.

L5-5: No.

6. Have you contacted Andersen Consulting to request assistance with the Concept Demo? If so, please comment on how prompt, adequate, and effective the response you received was.

L1-6: Andersen (specifically Bill Sasso and Mike DeBellis) have been more than willing to help. We greatly appreciate their help with acquiring CD, participating in the training sessions, as well as extending CD.

L2-6: Andersen provided us with the "reset-resource" fix.

L4-6: No need to request assistance, yet.

L5-6: No.

7. Please feel free to make any additional suggestions or comments regarding your experience with the Concept Demo.

L1-7: Please revise the manual to make it more readable. Also, provide a short summary of what each scenario is trying to do and what the starting point is and what the ending point is. As some scenarios have overlaps, unless this information is clearly stated, the user may be confused "repeating" the same steps from a previous scenario without a clear understanding of the concept being demonstrated. [paragraph] The CD has provided us with an opportunity to expose the Navy and other DoD officers in our Information Technology Management curriculum to the exciting field of KBSE. A demonstration accomplishes much more than what several lectures can not accomplish. We are glad to be able to bring awareness of the technology to the "right" audience. [paragraph] It has been a pleasure working with the system and Andersen. Look forward to more of the same ...

L2-7: I thought the Concept Demo showed its pieces well enough, but there were a lot of features and capabilities in the CD and I did not get a good sense of how they all played together. To some extent, the scenarios were disjoint. I did not attend the CD training session, so maybe I missed out on this, but I would have liked a level of description of the CD one level more abstract than the user's manual. I know there were a number of papers about the CD but these seem to have mixed the vision of the CD with the actual implementation. Maybe what I am looking for is a combination high-level design document for the CD (so that I could start to understand how the pieces fit together) and a CD methodology document. I am not sure these observations are completely fair to the CD as this information is probably available from a combination of sources, but still I had trouble seeing how everything fit together.

A.2 Feedback from those who have not completed the license process

Of the ten organizations currently in the licensing process, we received feedback from eight of them (80% response rate). These organizations are coded as follows:

Code	Organization Description	Refine?
N1	University Research Center (U.S.)	No
N2	University Research Center (U.S.)	No
N3	University Research Center (FDR)	No
N4	DoD Training Center	No
N5	DoD Research Agency	No
N6	University Research Center (U.K.)	No
N7	Industrial (Telecom) Technology Center (US)	No
N8	Non-Profit Research Institute (U.S.)	No

Two in-licensing-process organizations did not respond to the survey: a university computer science group (FDR) and a non-profit software engineering center (U.S.).

A representative of N5 attended the Concept Demo training program, but is no longer involved in that organization's Concept Demo acquisition efforts. None of the other in-license-process organizations have been represented at Concept Demo training programs.

1. Has your organization decided not to complete the KBSA Concept Demo licensing process, or is the process simply on "hold" for a while?

N1-1: ... I was the primary motivator for obtaining the system. I have since left N1 and am now at the University of X. The process is on hold for me and is probably dead at N1. I am proposing a new course in this department to deal specifically with knowledge-based software engineering, although it will be a year or so before I can teach it.

N2-1: We are on hold, pending availability of funds to acquire the necessary software licenses from KBSA-CD contributors (i.e., we don't have the money in any of our contracts at this time).

N3-1: ... we strongly plan to install the Concept Demo here, especially after I saw it in Monterey and X (on of my research associates) at KBSE. In fact, we are planning to use it in a class we teach jointly in the spring. The problem we was (a) the Refine license which took a while to clear, (b) a decision on whether to buy a full Common Lisp license or just go with the special offer (we have been Prolog users so far), and (c) the slow bureaucracy in our university which has to process the official orders. But it will come!

N4-1: On hold, due to need for purchase of enabling software.

N5-1: For reasons I don't fully understand, negotiations concerning licensing (which I understand needed to be completed prior to demo) were never completed. The bottom line is that none of [us] here have seen the demo.

N8-1. The process is "on hold" due to the need to acquire Software Refinery.

2. If you have decided not to complete the licensing process, please describe any factors that led to that decision, especially in terms of any aspects that you believe Andersen or Rome Lab could modify.

N1-2: The decision not to proceed at N1 was motivated primarily by the need for the Refine license. Partially cost and partially just having to deal with one more group. At this point, I feel it is a dead issue at N1 because no one there is as interested in KBSE as I was.

N6-2: Our interest in the Concept Demo was due to the fact that we are doing research into automated software engineering tools and we wanted to see what the Concept Demo was all about in relation to our work. When it became clear that it required other pieces of software like Refine, which we weren't interested in buying for a one-off demo, it began to sound like much more trouble than it was worth to check it out.

N7-2: When I originally looked at the specs for KBSA, I saw another license (I forget for what) which we would have had to procure for a nontrivial amount of money. I did not see a business case which I could make for spending that mount, so I didn't proceed with licensing KBSA.

3. If the process is simply on "hold," please describe the nature or causes of the hold, especially any of those causes that you believe Andersen or Rome Lab could modify.

N1-3: As far as my work at the University of X is concerned, things are on hold until I get settled in. As I noted earlier, I want to do a class on KBSE techniques, but must get the course approved and on the books. This should not be a problem, but will take some time. My current research area involves case-based reasoning and formal specifications, thus I consider the Concept Demo important to my research as well. Again, it will just take some time before I get started back up here.

N2-3: At this time, there is interest at the School of Business and at the CS dept at N2 to get a copy of KBSA-CD for experimentation. If the KBSA-CD package (including required platform software, e.g., Franz CL needed to run Kestrel/Reasoning Systems software) were available to universities at

"no cost" we would seek to get a copy immediately. But these days, we do not have the approximate \$1000 in software license money at our disposal.

N8-3. We were informed by Kestrel that we were not eligible for an educational Refine license, and so have requested funding for a limited use license in next year's budget.

Andersen Consulting is aware that at least one large aerospace industry computer services organization regards the "Additional Consideration" clause in the Concept Demo license as overly restrictive. This clause requires that — if Andersen so requests — extensions to the Concept Demo made by the licensee be licensed to Andersen under terms similar to the terms of the Concept Demo license. When they pointed this out to us, we responded that this clause was present in licenses for several other KBSA prototypes (which we believe they have licensed), and suggested that they propose alternate language. To date, they have not responded with a proposal for alternative language.

4. Have you discussed any of these causes/factors with anyone at Andersen Consulting? If so, how would you characterize the response?

N1-4: I have had no contact with Andersen Consulting.

N2-4: We have had a number of interesting discussions with friends at AC to try to identify some collaborative projects whose funding would resolve this resource constraint.

N8-4. Most of my interaction has been with Andersen's Mike DeBellis; he has been helpful and generous with his time.

5. Please feel free to make any additional suggestions or comments regarding your experience with the Concept Demo.

N1-5: I will obtain the Concept Demo if it is at all possible for me to do so, both for my research and for teaching.

**MISSION
OF
ROME LABORATORY**

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C³I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.